

100

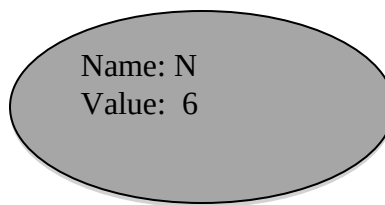


Fig.1

200

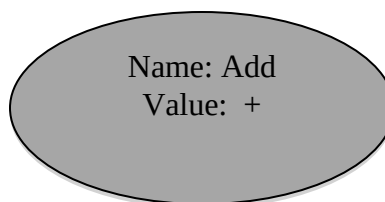


Fig. 2

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

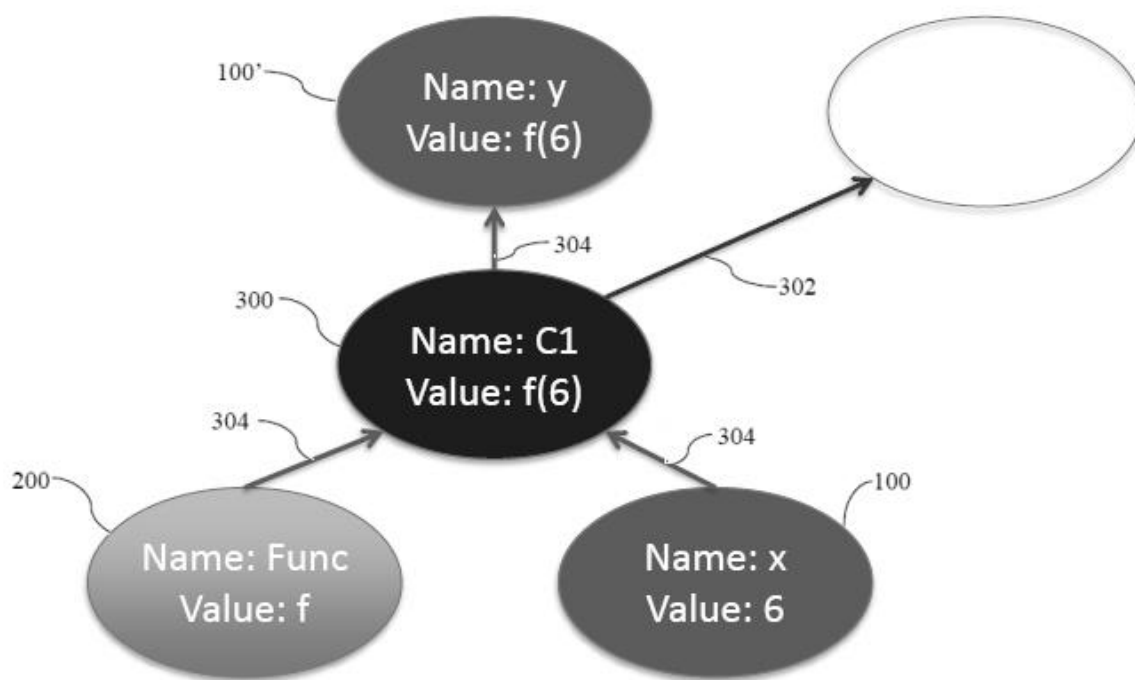


Fig. 3

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

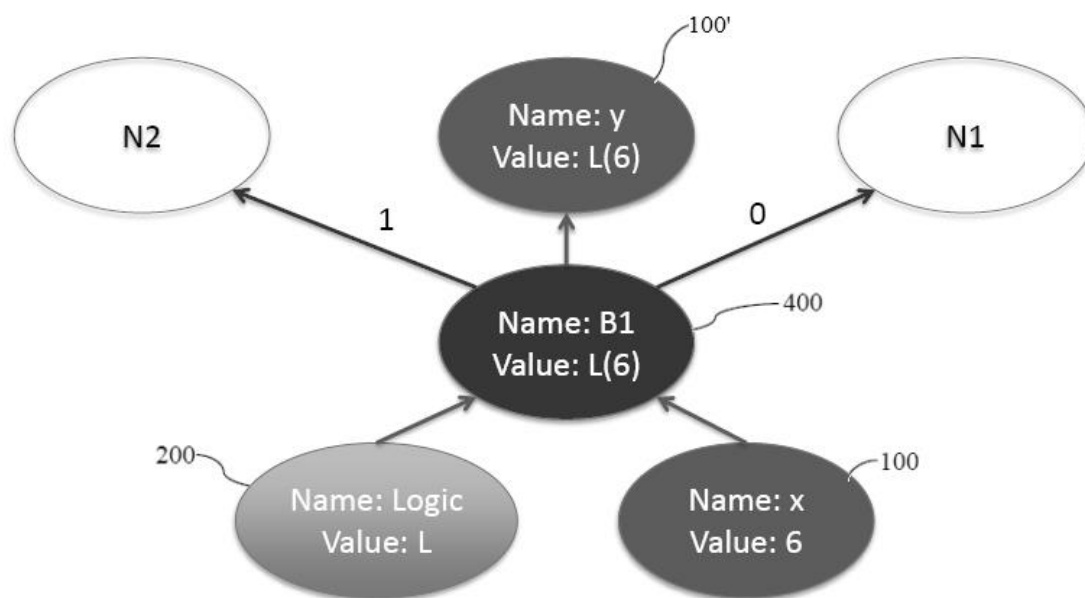


Fig. 4

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

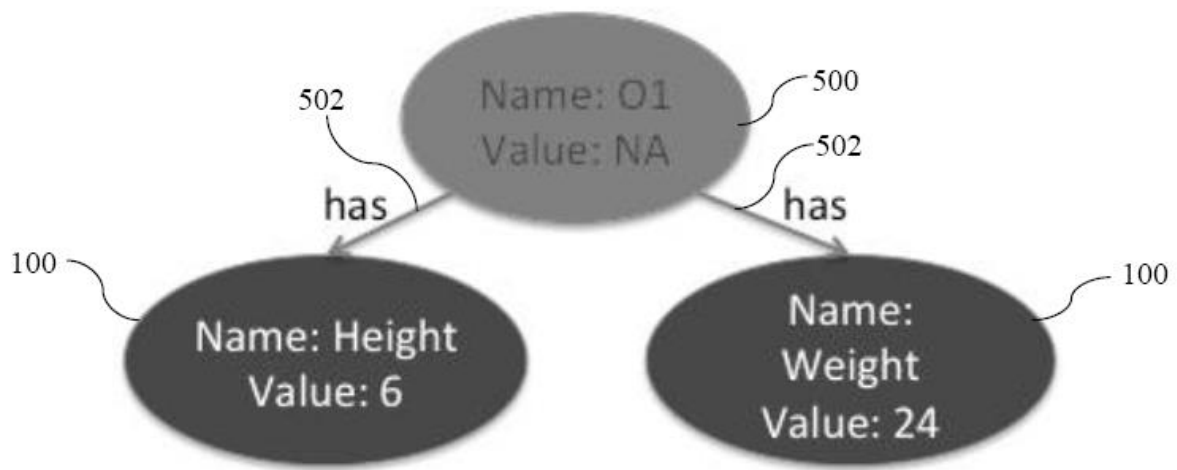


Fig. 5

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

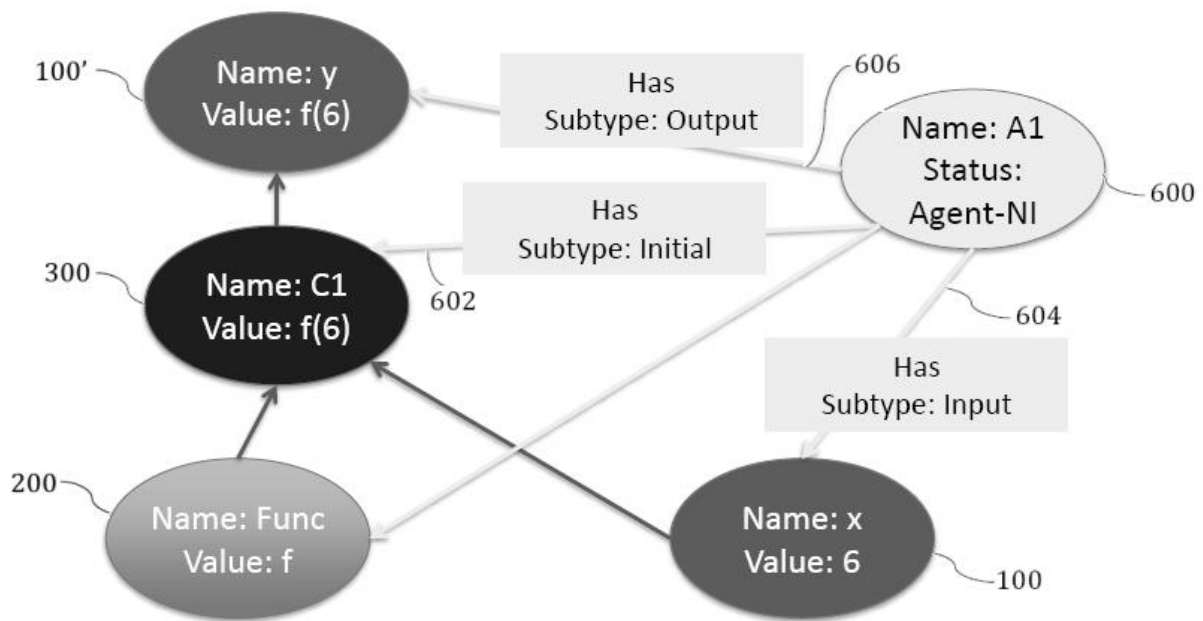


Fig. 6

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

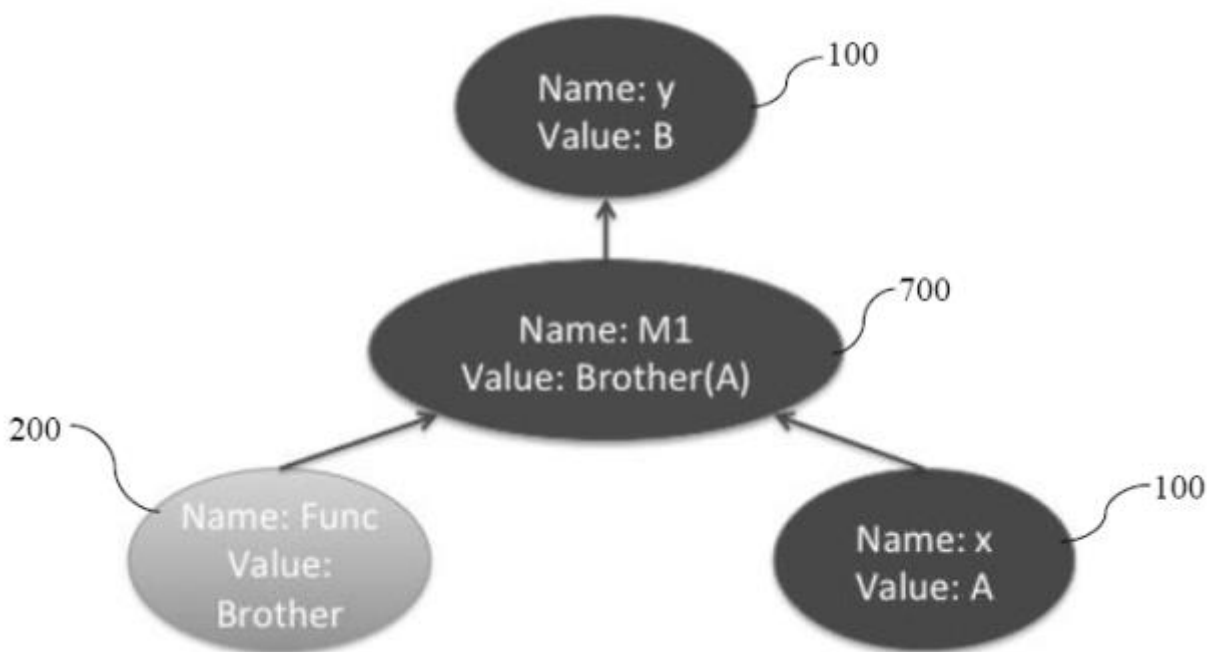


Fig. 7

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

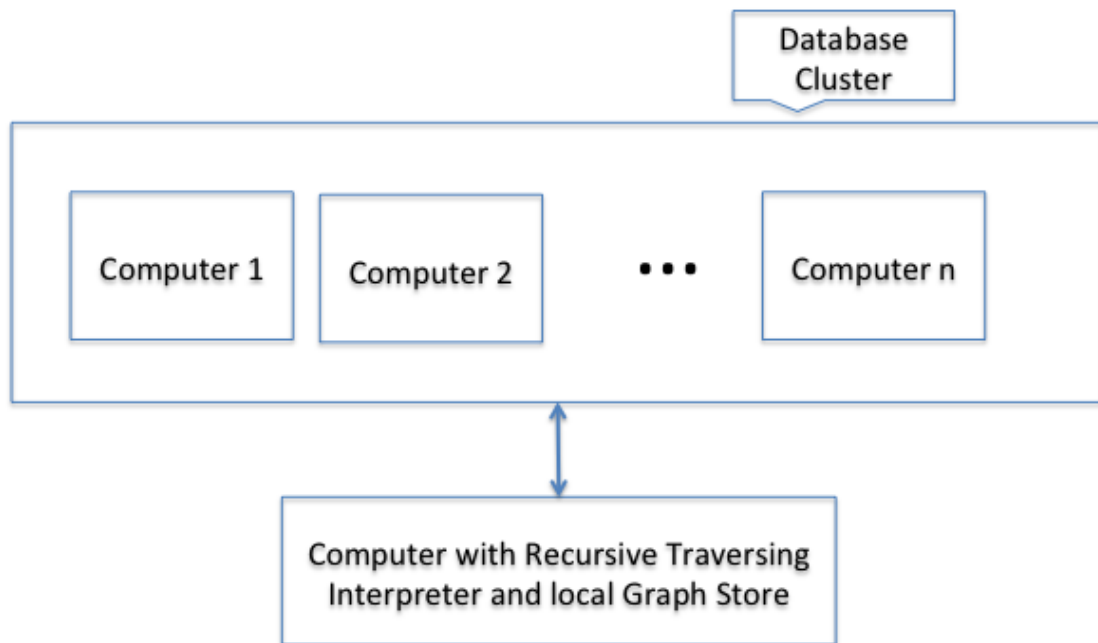


Fig. 8

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

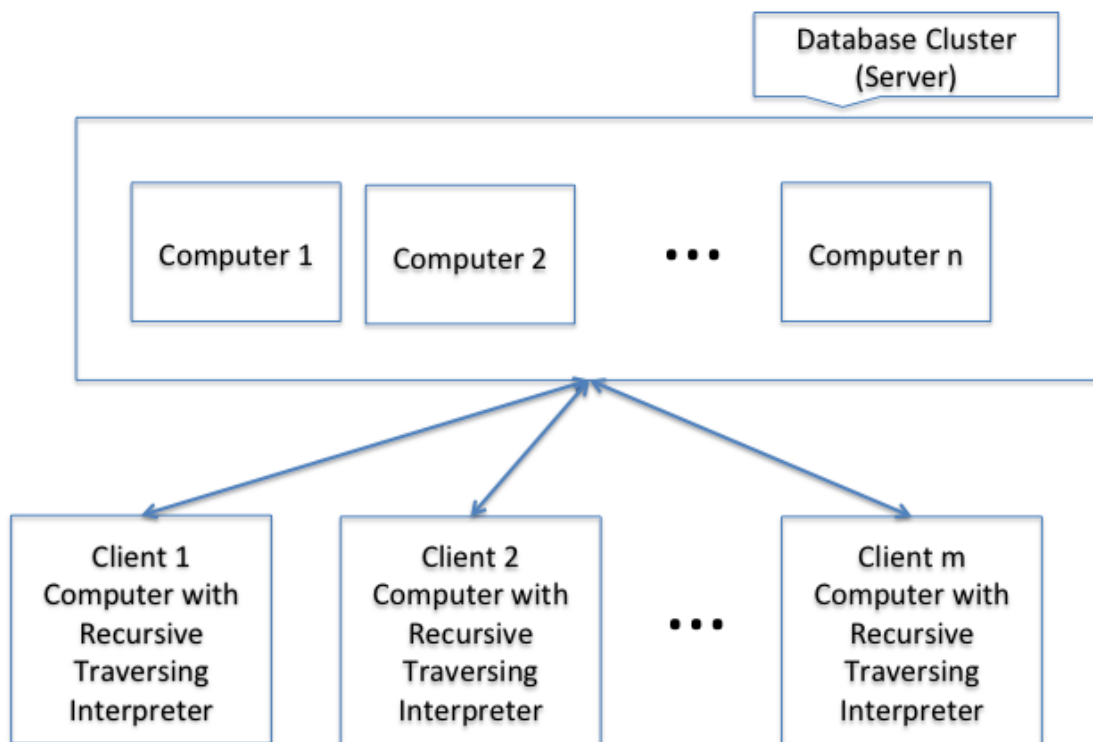


Fig. 9

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

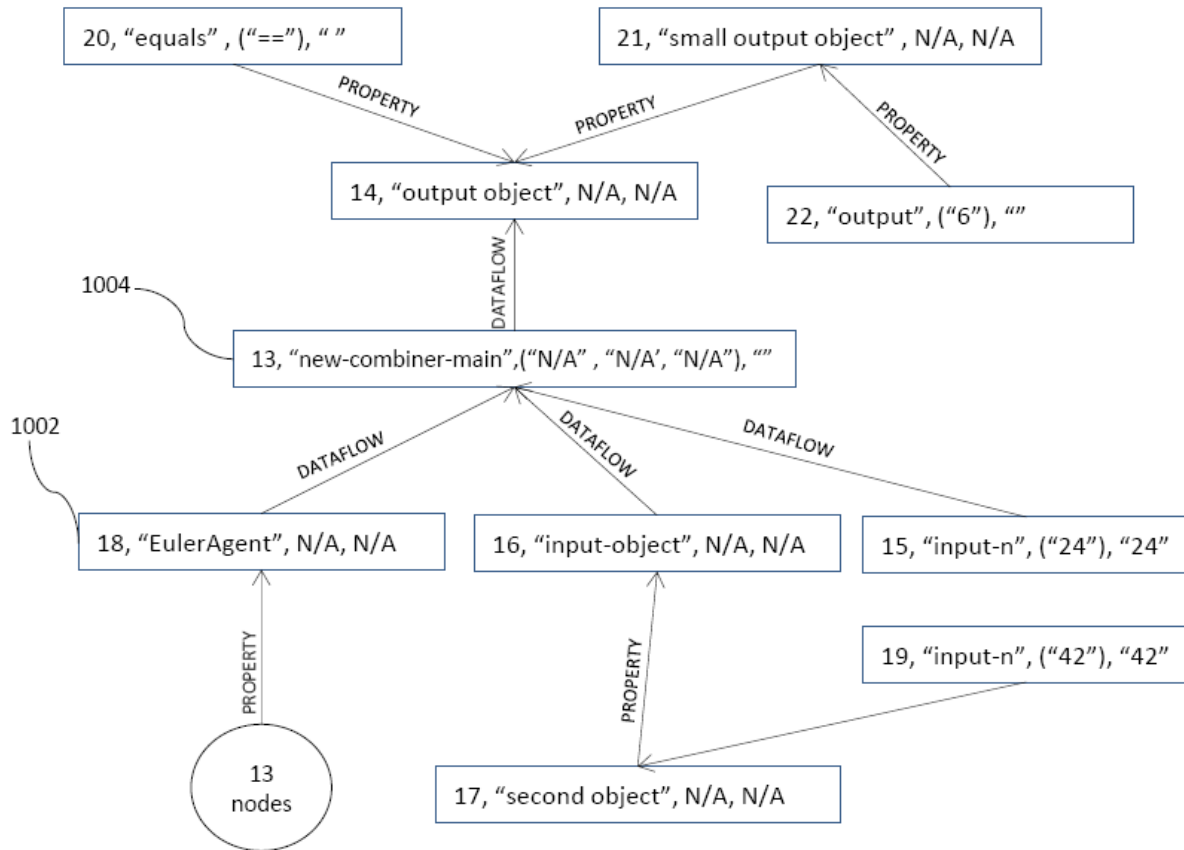


Fig. 10

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

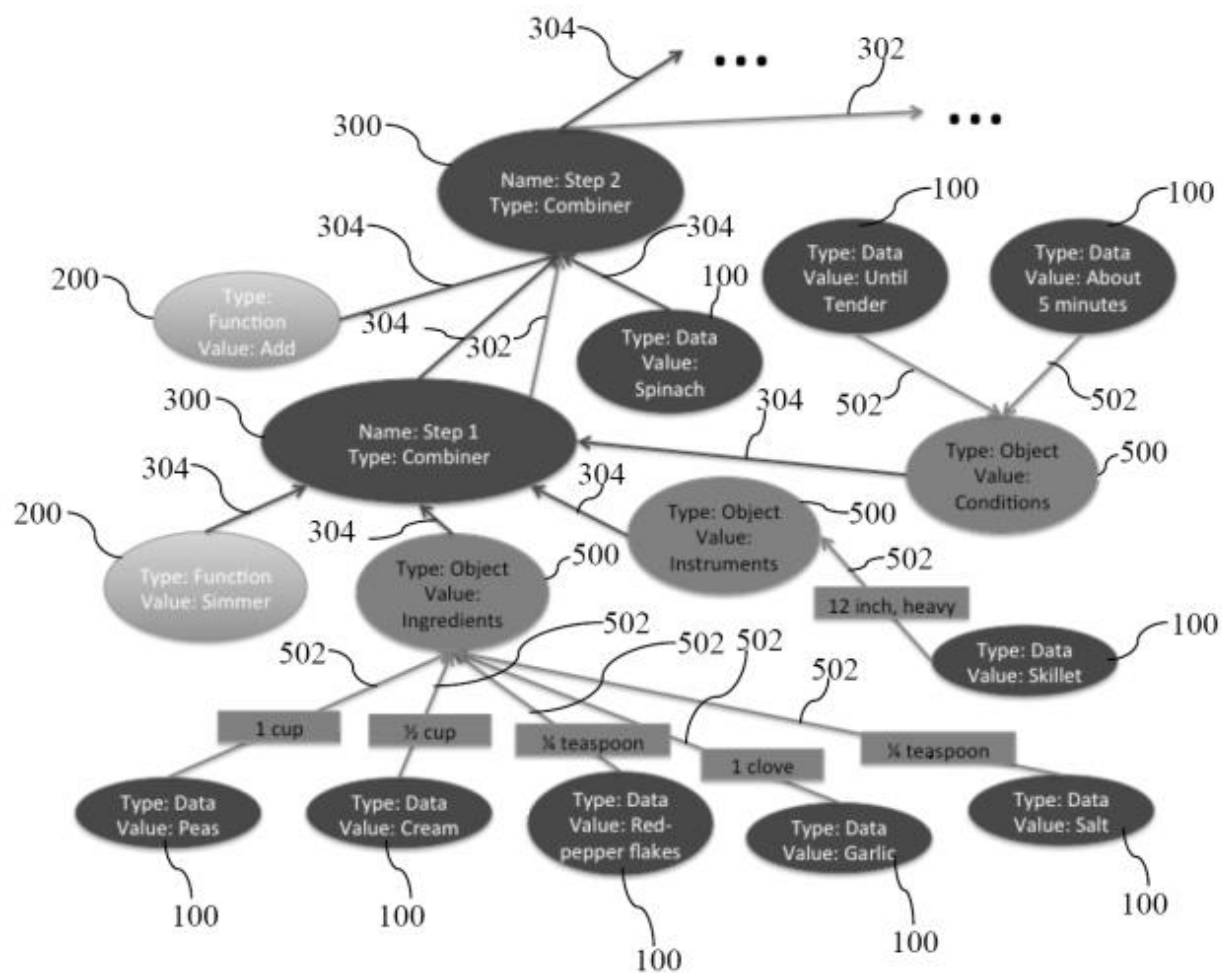


Fig. 11

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

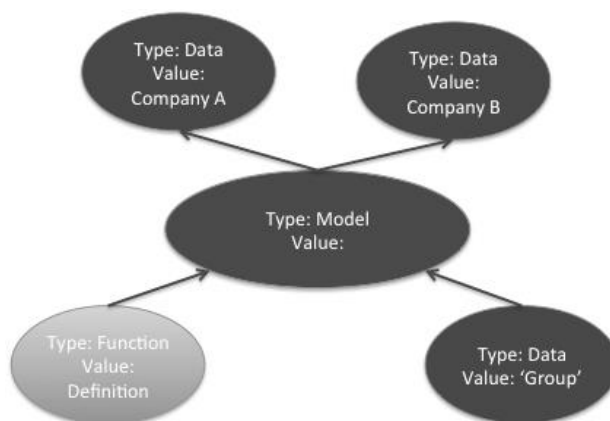


Fig. 12

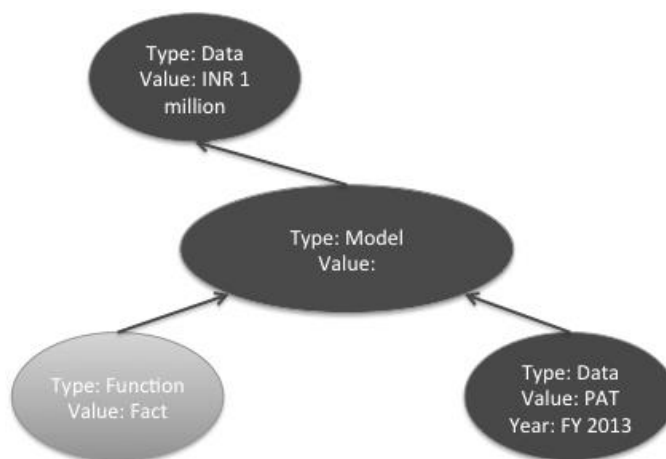


Fig. 13

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

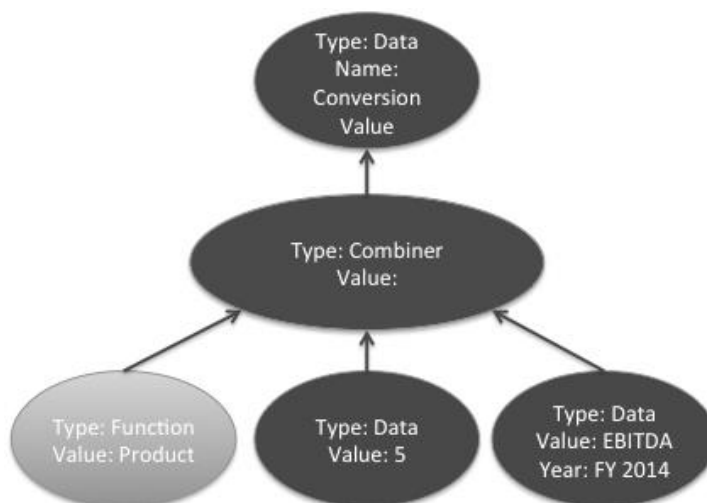


Fig. 14

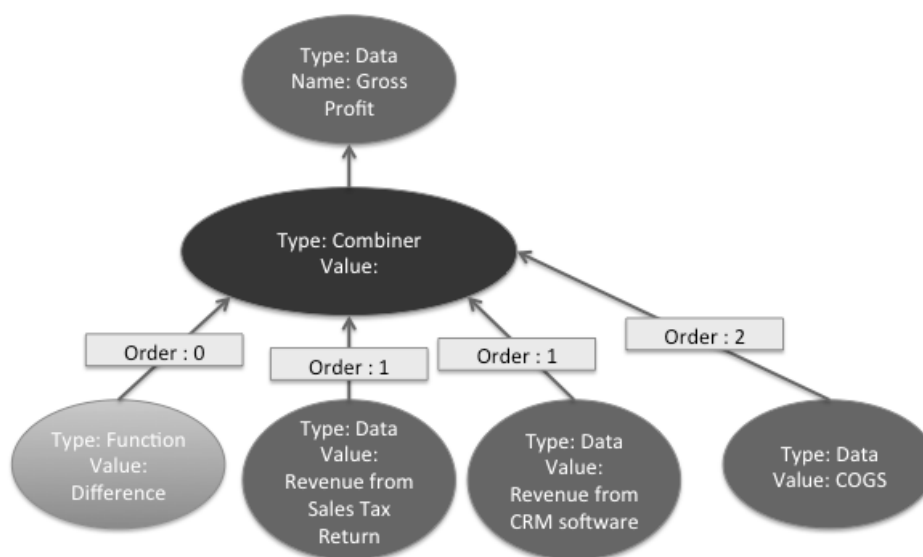


Fig. 15

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

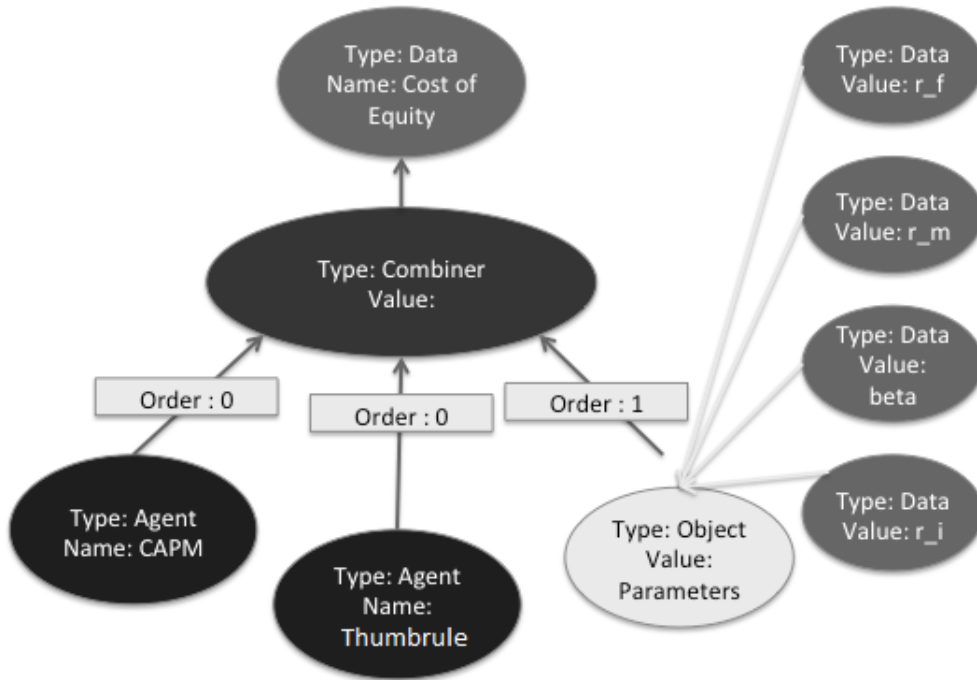


Fig. 16

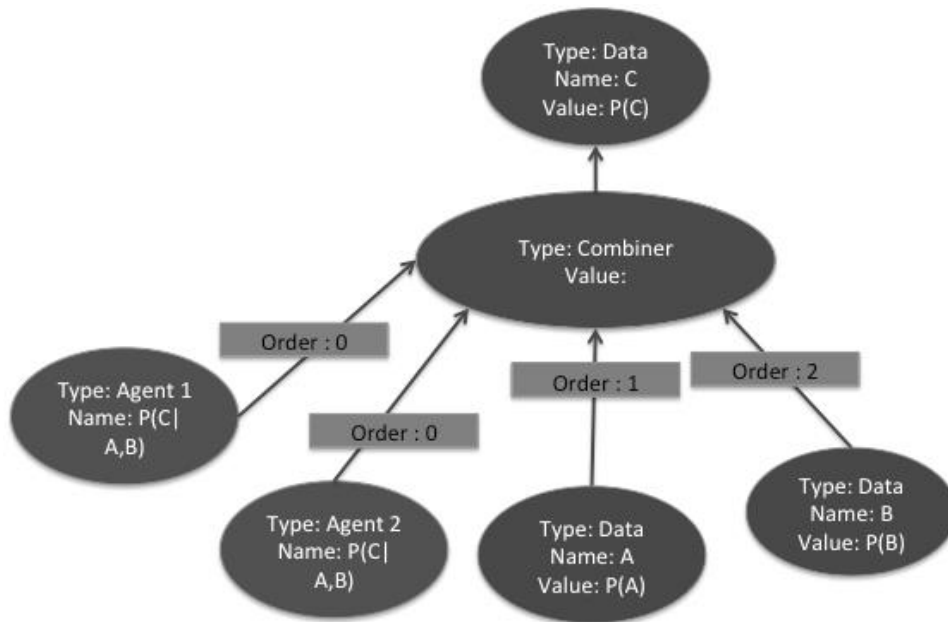


Fig. 17

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

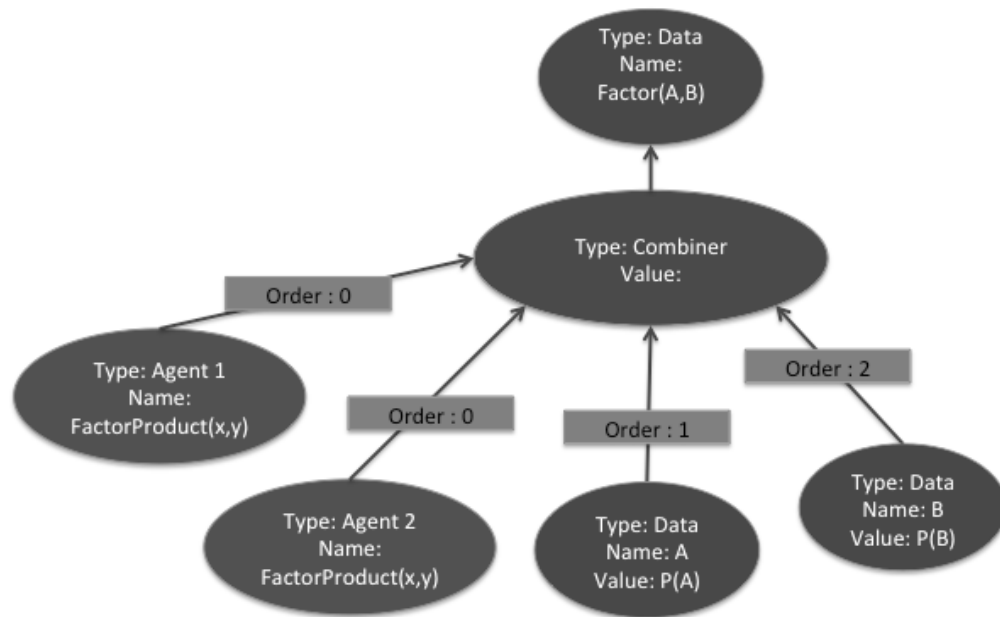


Fig. 18

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

ABSTRACT

GRAPH BASED ONTOLOGY MODELING SYSTEM

A graph based ontology modeling system comprising a knowledge base server containing information in the form of graph comprising a plurality of nodes and a plurality of edges; a client system having a recursive-traversing interpreter (RTI) enabling queries on said graph using a combination of eager and lazy evaluation method and updating values of various nodes in said graph; wherein said graph comprises: dataflow edges; controlflow edges; property edges; data nodes; function nodes; combiner nodes; branch nodes; agent nodes; and model nodes; wherein based on query of a user specifying the starting node for traversal, RTI updates the values of various nodes.

Fig. 6

100

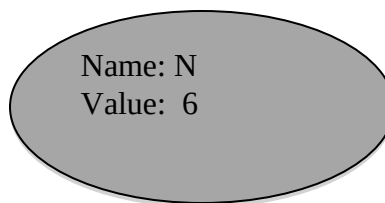


Fig.1

200

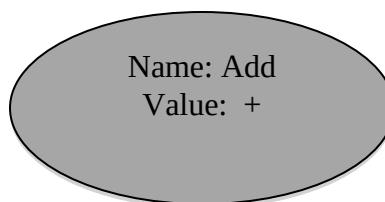


Fig. 2

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

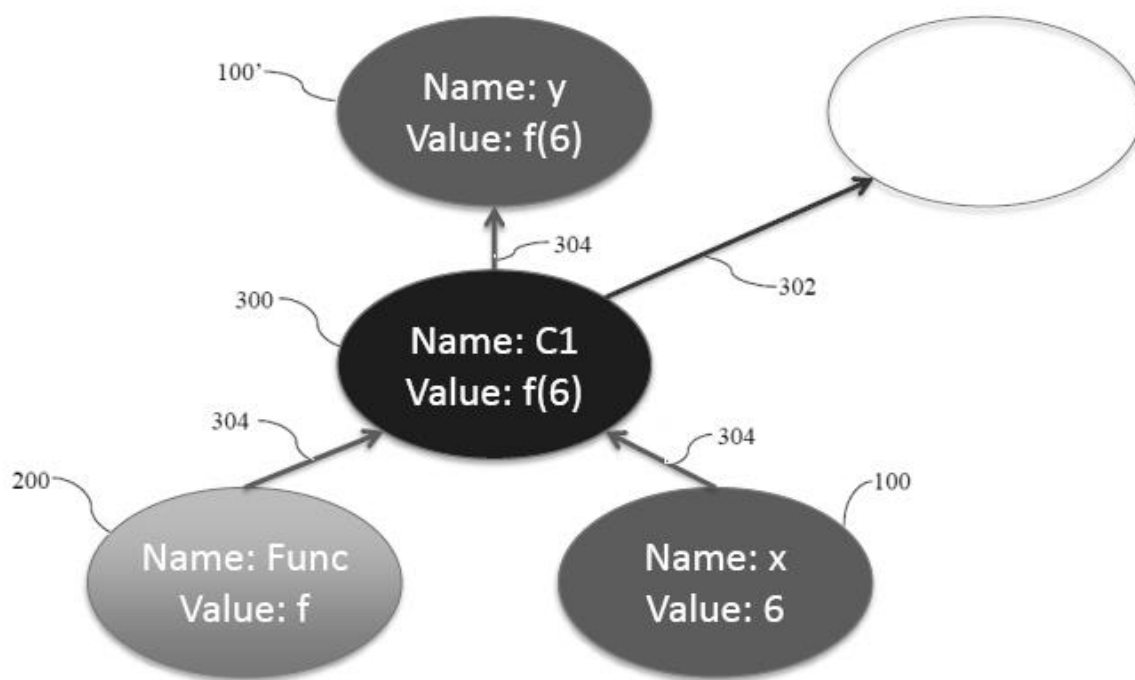


Fig. 3

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

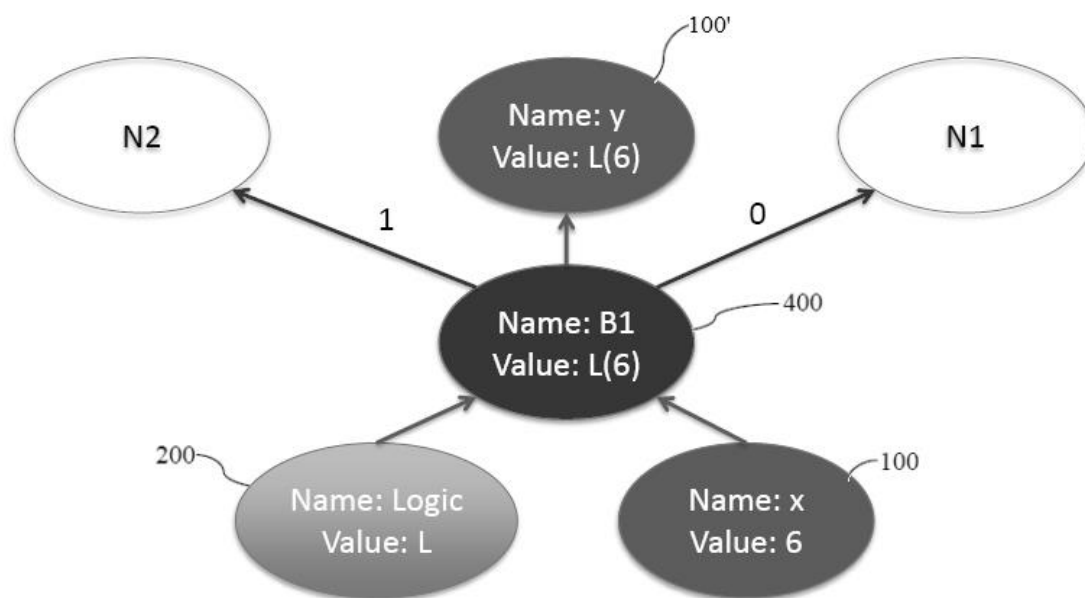


Fig. 4

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

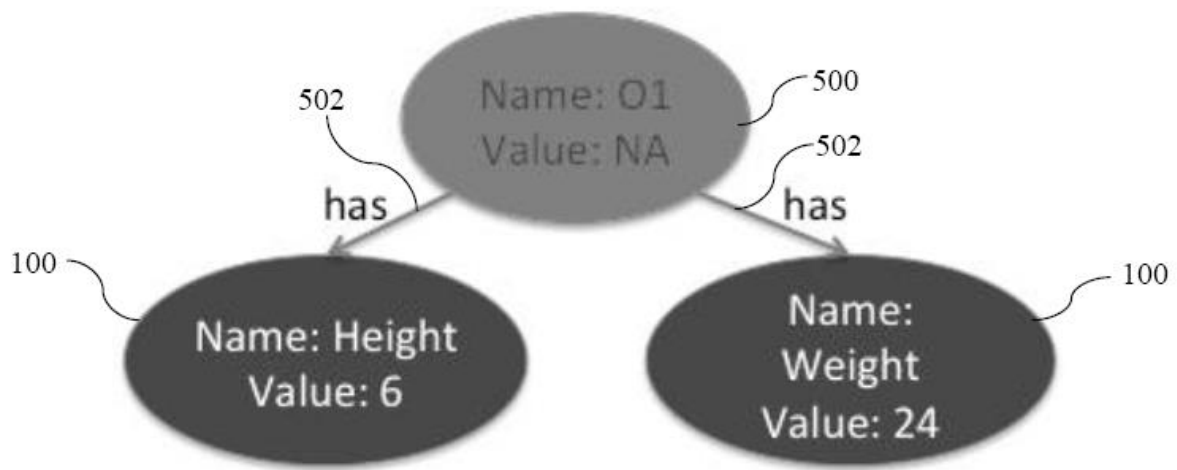


Fig. 5

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

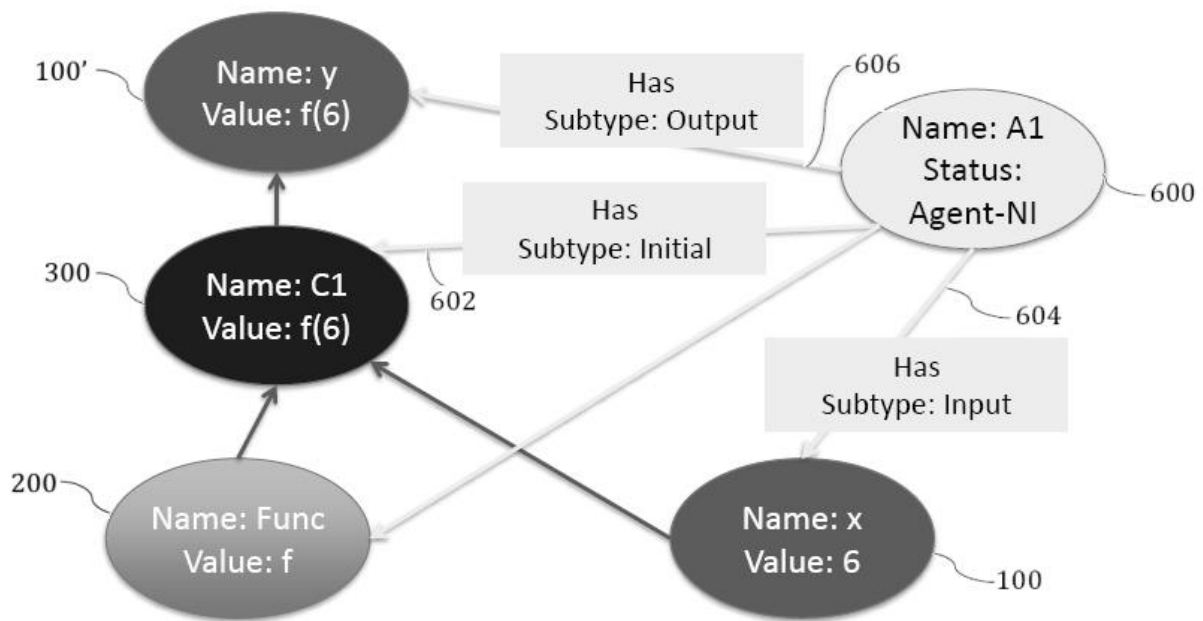


Fig. 6

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

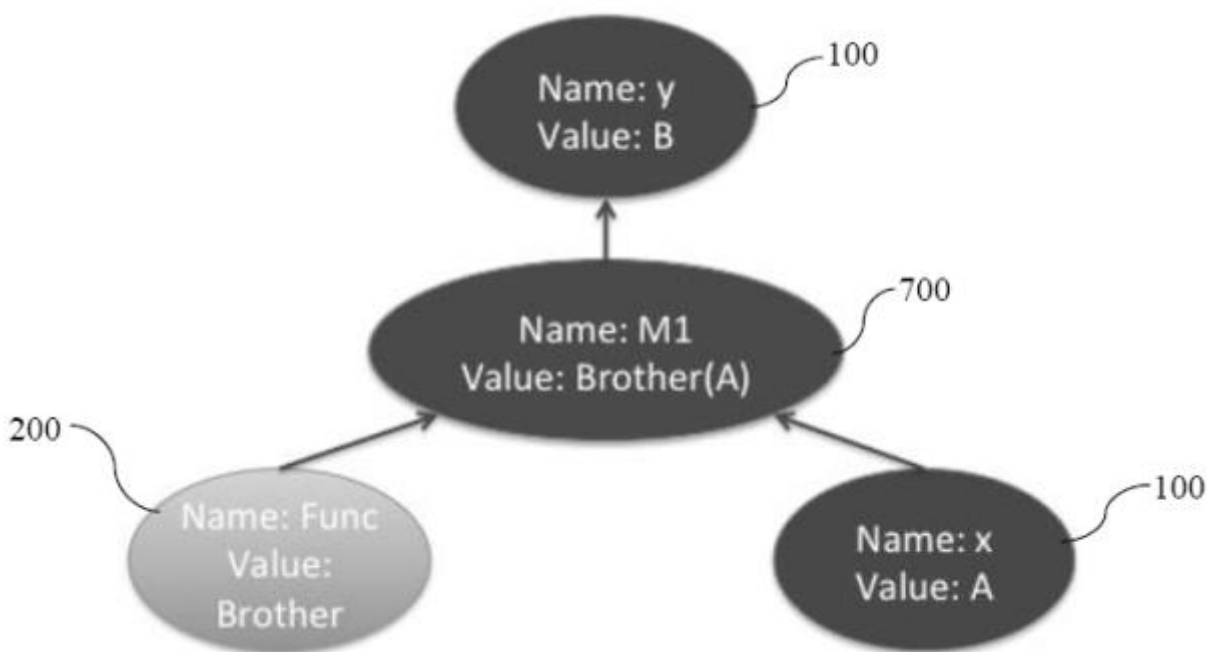


Fig. 7

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

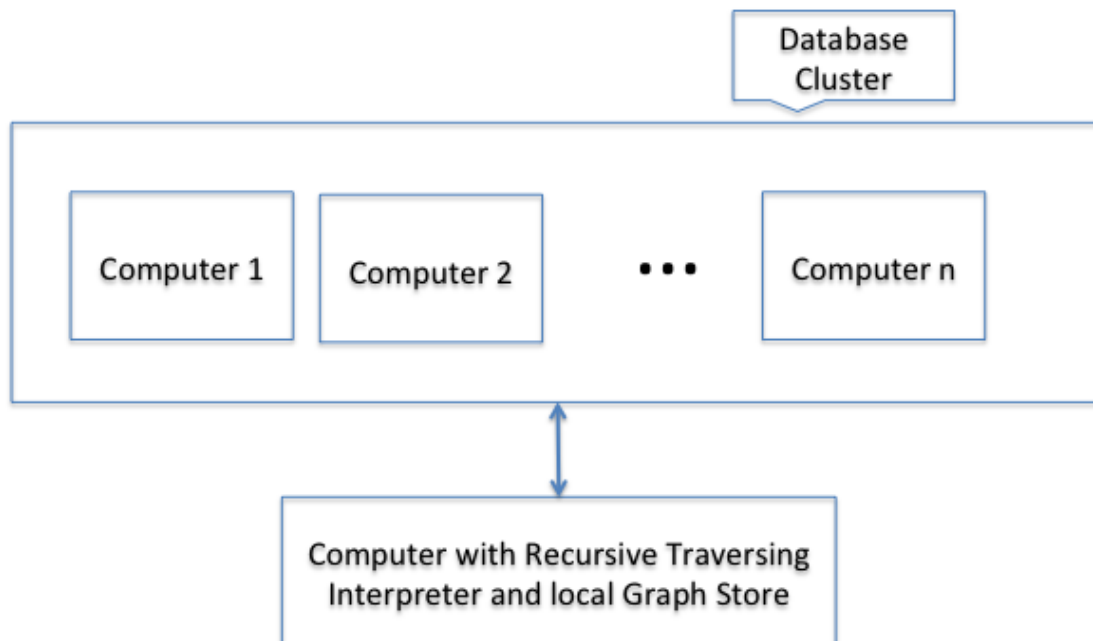


Fig. 8

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

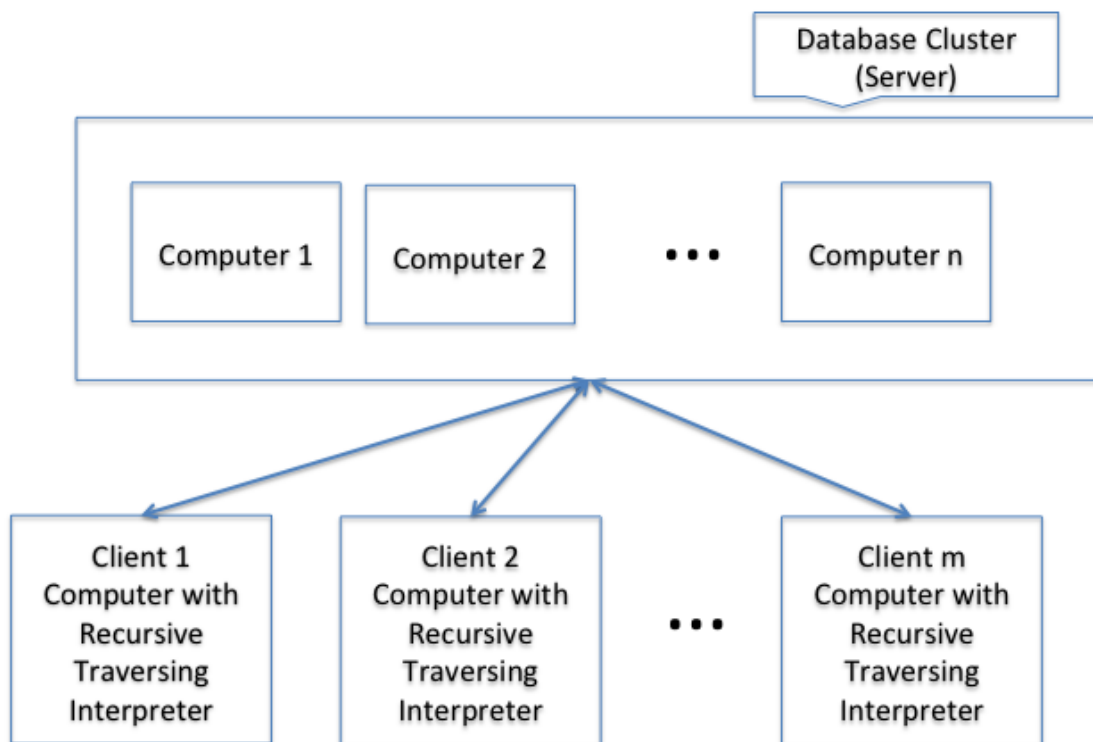
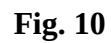


Fig. 9

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471



(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

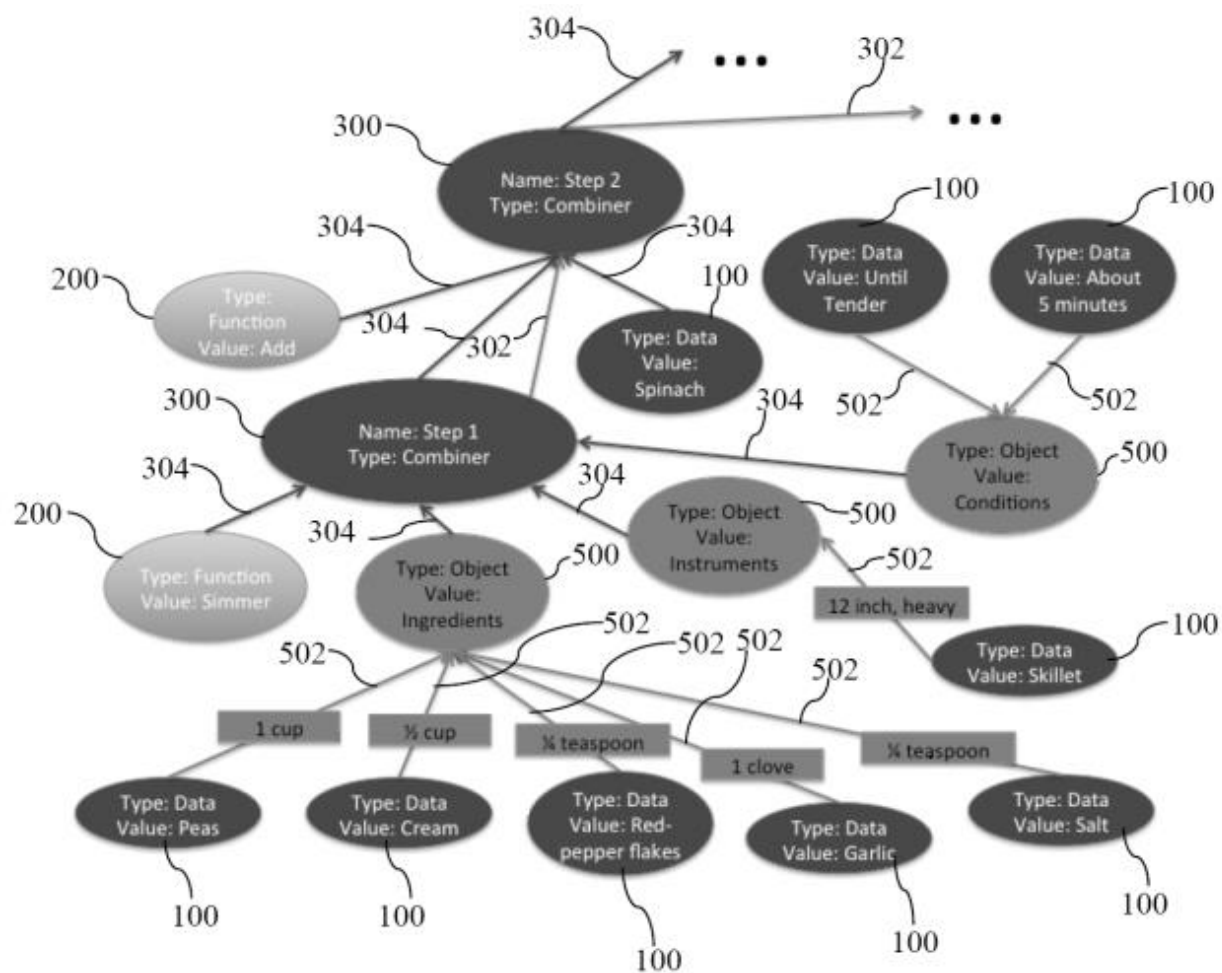


Fig. 11

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

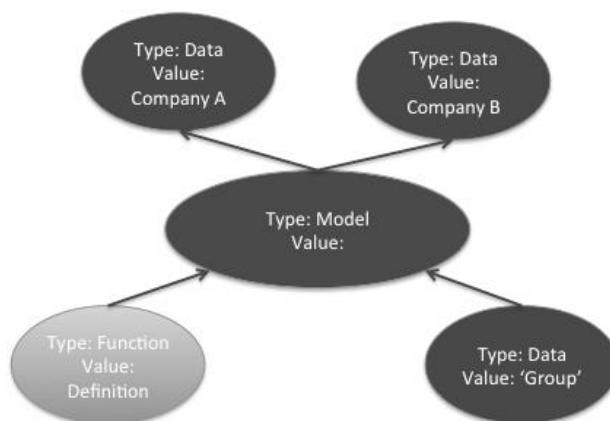


Fig. 12

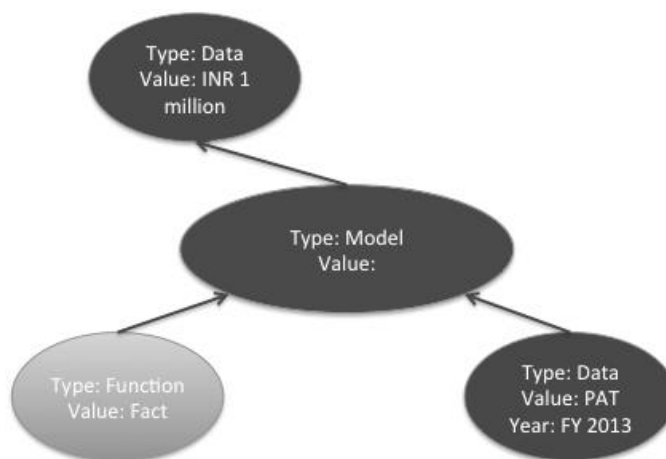


Fig. 13

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

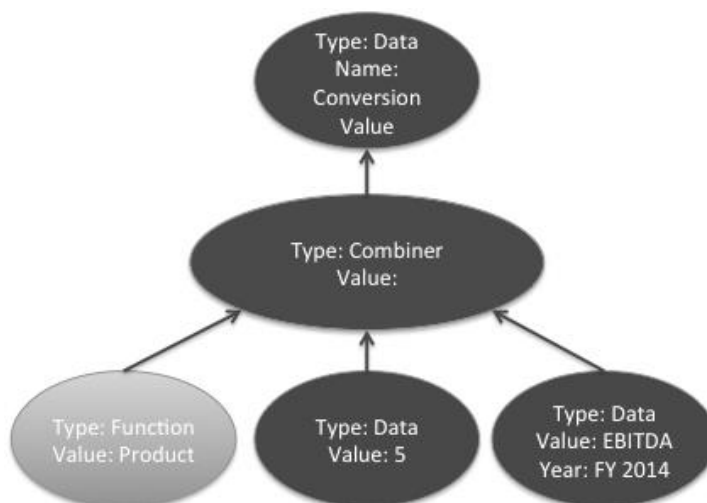


Fig. 14

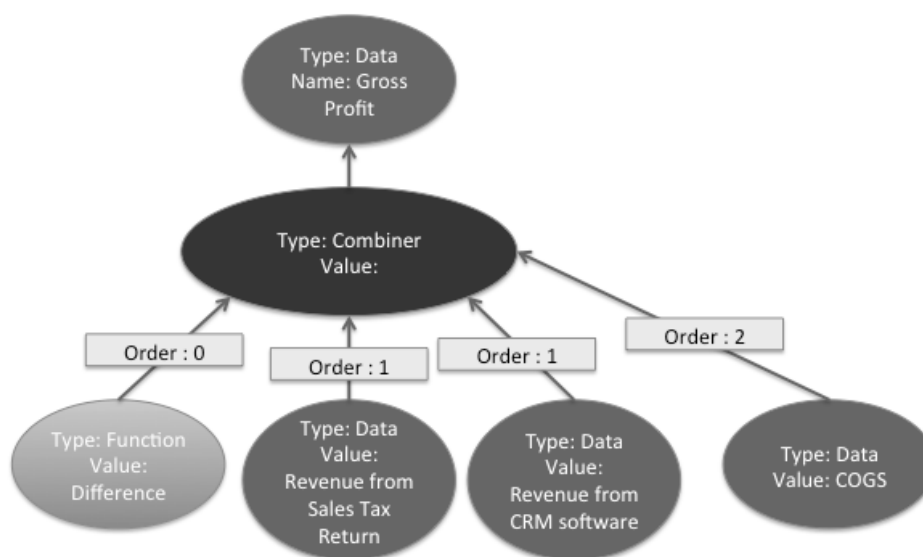


Fig. 15

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

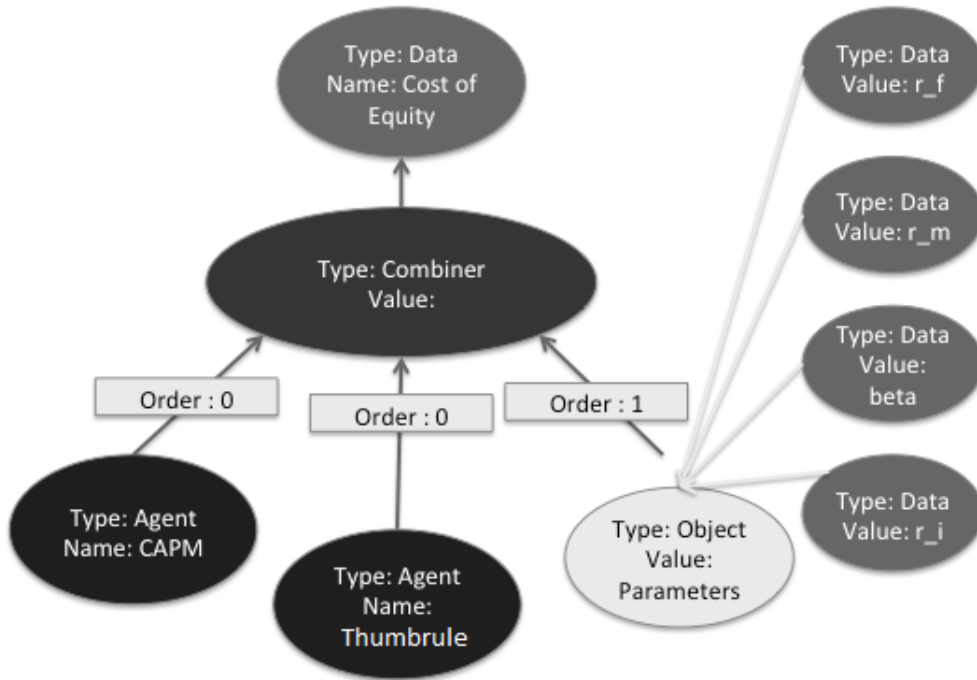


Fig. 16

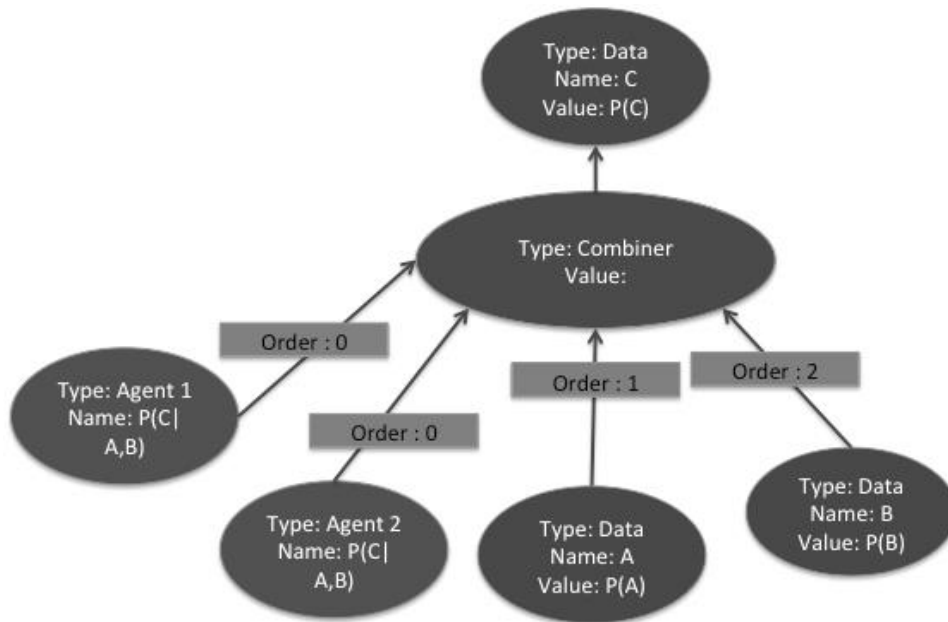


Fig. 17

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

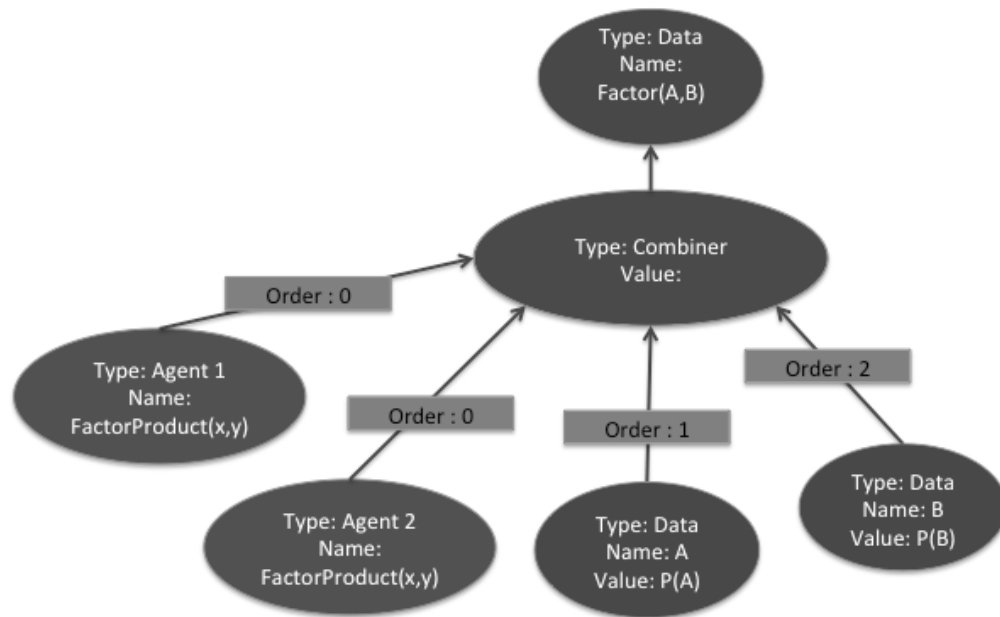


Fig. 18

Jayakumar Subramanian
By his Agent

(Nikhil Ranjan)
of Khaitan & Co
Reg No IN/PA-1471

ABSTRACT

GRAPH BASED ONTOLOGY MODELING SYSTEM

A graph based ontology modeling system comprising a knowledge base server containing information in the form of graph comprising a plurality of nodes and a plurality of edges; a client system having a recursive-traversing interpreter (RTI) enabling queries on said graph using a combination of eager and lazy evaluation method and updating values of various nodes in said graph; wherein said graph comprises: dataflow edges; controlflow edges; property edges; data nodes; function nodes; combiner nodes; branch nodes; agent nodes; and model nodes; wherein based on query of a user specifying the starting node for traversal, RTI updates the values of various nodes.

Fig. 6

FIELD OF THE INVENTION

This invention relates to graph based ontology modeling system.

BACKGROUND OF THE INVENTION

An ontology designates a data model that represents domain knowledge and is used to query and reason about the properties of the objects in that domain and the relations between them. The syntax and semantics of the specific ontology language used govern the expressiveness and utility of the language to model knowledge and derive new knowledge. Traditionally, ontologies have used a declarative language with first order logic like semantics along with a reasoning/inference system to handle queries. At present, most of these systems use Resource Description Framework (RDF) or similar approaches at their core. Since ontologies essentially capture relationship between entities, they have evolved from a textual representation to a graph based representation with the same underlying characteristics. With the advent of graph databases, it is now easier to store graphs directly instead of converting them into relational tables. This also helps in scaling as most queries in graphs can be viewed as graph traversals, which allow replacing costly join operations in equivalent relational algebra tables. However, this declarative language based approach is not very efficient when an ontology is to be used for dynamical computations like updating the state of a system with many agents and objects. For this purpose, imperative or functional language characteristics would prove useful. This has been explored to some extent in the system called OpenCog though for a different purpose. This system uses a hypergraph database and to specify functions it uses a combinator-based approach. The disadvantage of such approaches is that for evaluating recursive functions using only graphs,

they require creation of new nodes in the graph for every recursion instance and thus would run into scalability issues.

Another approach used for implementing an imperative approach is the graph transformation approach which has been discussed in “Graph Transformation in a Nutshell, Reiko Hackel, Electronic Notes in Theoretical Computer Science 148 (2006) 187-198”. In this, rules are used to transform the graph database. Typically, these rules have a left-hand side and a right-hand side, where both are subgraphs. The left-hand side of a rule is matched with the nodes in the graph to identify a subgraph that has the same pattern. This subgraph is then replaced with the subgraph specified by the right-hand side of the rule. Some of the disadvantages in this and similar graph transformation approaches is that they require an expensive pattern-matching step (a structural step) that does not scale well as the graph scales. The OpenCog system has written its foundations on a hypergraph re-writing system (a type of Graph Transformation).

Graph based functional and imperative approaches have been used primarily in the form of visual programming languages and not for ontological engineering.

US 2005/0097561 discloses a system and method for managing data, such as in a data warehousing, analysis, or similar applications, where dataflow graphs are expressed as reusable map components, at least some of which are selected from a library of components, and map components are assembled to create an integrated dataflow application. Composite map components encapsulate a dataflow pattern using other maps as subcomponents. Ports are used as link points to assemble map components and are hierarchical and composite allowing ports to

contain other ports. The dataflow application may be executed in a parallel processing environment by recognizing the linked data processes within the map components and assigning threads to the linked data processes.

US 7299458 discloses a method of forming a control-dataflow graph that includes separating a control flow graph into two or more basic blocks, and converting said two or more basic blocks into code blocks, where the code blocks are formed into the control-dataflow graph. Another embodiment of the invention includes a method of forming a control-dataflow graph that includes separating a control flow graph into two or more basic blocks, forming a lode node in at least one of said basic blocks, forming a store node in at least one of said code blocks, inserting a delay node in at least one of said code blocks, segregating external hardware logic modules from said control flow graph, and converting said two or more basic blocks into code blocks, wherein the code blocks are formed into the control-dataflow graph.

US 7703085 describes a system and method for compiling computer code written to conform to a high-level language standard to generate a unified executable containing the hardware logic for a reconfigurable processor, the instructions for a traditional processor (instruction processor), and the associated support code for managing execution on a hybrid hardware platform. Explicit knowledge of writing hardware-level design code is not required since the problem can be represented in a high-level language syntax. A top-level driver invokes a standard-conforming compiler that provides syntactic and semantic analysis. The driver invokes a compilation phase that translates the CFG representation being generated into a hybrid controlflow-dataflow graph representation representing optimized pipelined logic which may be processed into a hardware

description representation. The driver invokes a hardware description language (HDL) compiler to produce a netlist file that can be used to start the place-and-route compilation needed to produce a bitstream for the reconfigurable computer. The programming environment then provides support for taking the output from the compilation driver and combining all the necessary components together to produce a unified executable capable of running on both the instruction processor and reconfigurable processor.

US 7316001 describes a software system including an Object Process Graph for defining applications and a Dynamic Graph Interpreter that interprets Object Process Graphs. An Object Process Graph defines all of an application's manipulations and processing steps and all of the application's data. An Object Process Graph is dynamic, making it possible to change any aspect of an application's data entry, processing or information display at any time. When an Object Process Graph is interpreted, it functions to accept data, process the data and produce information output. Modifications made to an Object Process Graph while it is being interpreted take effect immediately and can be saved. Object Process Graphs and Dynamic Graph Interpreters can be deployed on single user workstation computers or on distributed processing environments where central servers store Object Process Graphs and run Dynamic Graph Interpreters, and workstation computers access the servers via the intranet or local intranets.

In all the aforementioned approaches, the emphasis is on graphical specification of a programming language and hence they have very limited expressive power to handle meta-queries (i.e. structural queries) as also in modeling complex agents, objects and their interactions. In a Dataflow Graph, as explained in US 2005/0097561, the nodes of the graph are data

transformation processes that have input and output ports. The edges are data pipes that connect output ports to input ports. There are also parent to child edges denoting component ownership. These graphs could either be flat graphs or composite graphs. The execution of a dataflow graph is done by producing an execution plan, which is an internal and private data structure. This data structure is a flat (non-hierarchical) data structure. All the structural information of the graph, which are important for human understanding, but are not needed for execution, are deleted while creating the execution plan.

In US 7299458, High Level Programs are converted into executable instructions for different hardware execution platforms using Control-Dataflow Graphs. In this approach, a program in a high-level programming language is converted into a Control flow graph, which is then converted into a Control-Dataflow graph. This is then partitioned into a reconfigurable hardware portion and an instruction processor portion. The reconfigurable hardware portion is then converted into a hardware definition language file, which is then ultimately used to create the unified executable after some processing. This patent also describes use of Multi-Adaptive Processors (MAPs) in the reconfigurable hardware portion. MAPs integrate Field Programmable Gate Array (FPGA) reconfigurable circuitry with logic to control the FPGAs and communicate with the instruction processor portion of the system. These MAPs may also be linked to instruction processors through switches/network adapter ports.

The first two types (RDF based and Graph Transformation based) of approaches described previously do not exploit the local structure of the graph for imperative computations. The third type of approach, though not an ontology, suffers from drawbacks in expressive power as well as in querying ability. To address these limitations, an approach is required that utilizes local

structure as well as uses a recursive traversal based approach for computation (or interpretation). This recursive traversal provides better expressive power as well as querying ability. The other key benefit provided by a graph structure that has not been used in the past is the power of traversal to relax the API (Application Programming Interface) specification requirements used in imperative approaches. Besides, the ability to express declarative structures in imperative form, finding the inverse of imperative specifications (at least in some cases), utilizing structural properties such as shortest path to construct new imperative structures, optimization and using structural properties for transferring knowledge between different domains are requirements that have not been explored in the prior approaches.

The present invention is contrived in consideration of the circumstances mentioned hereinbefore, and is intended to provide an ontology that has imperative as well as declarative characteristics including recursive-traversing interpreter (or RTI or Recursive Traversing Interpreter) that enables running imperative queries on this ontology. The implementation of both the Ontology and the recursive-traversing interpreter in a single processor based computer system is described. Its extension to multiple computers connected over a network to enable both representing 'big data' as well as parallel processing is described.

DETAILED DESCRIPTION

The invention may be better understood and further advantages and uses thereof more readily apparent when considered in view of the following detailed description of exemplary embodiments, taken with the accompanying drawings. These embodiments describe only a few of the various ways in which the principles of various other embodiments may be realized and

the described embodiments are intended to include all such embodiments and their equivalents and the reference numerals used in the accompanying drawings correspond to the like elements throughout the description.

In the Ontology specified in the present invention, seven types of nodes and three types of edges form the basic elements that provide imperative and declarative expressive power. The Node types have been defined loosely based on recursive function theory. In recursive function theory, the building blocks can be simplified as comprising of a given function, the principle of composition of functions and the principle of primitive recursion. In most modern functional programming languages based on lambda calculus (an alternate system to recursive function theory, but shown to be equivalent by Church), composition is achieved through an 'eval' operation applied to a set of elements, each of which are a part of the basic functions or data types described in the programming language. Recursion was achieved through the ability of a function to call itself. In imperative programming languages, loops serve the same purpose, though recursion is also permitted to increase expressive power, although it generally is the more inefficient approach. In the present invention, composition is represented by two types of Nodes (called Combiner and Branch in this invention) and recursion is achieved through looping using the fact that the two types of Nodes representing composition permit two different types of traversal behaviour. Two more types of Nodes are used to represent data and functions (called Data and Function in this work), which are the building blocks or Atoms as referred to in this specification. The final three types of Nodes are used for declarative specifications such as constructing compound entities from the Atoms mentioned above and the nodes that serve to represent composition (these are called Object, Agent and Model in this invention). Another kind

of declarative specification refers to representing declarative relations in the form of functions and one of the final three types of Nodes is used in such cases. In the present specification, Compounds refer to all Nodes that are not Atoms, that is, they are either Combiners, Branches, Agents, Objects or Models.

Referring to **Figs. 1 and 2**, an exemplary embodiment of node 'Atom' is shown according to present invention with only partial list of properties. There are two kinds of 'Atoms' – Data **100**, as shown in **Fig. 1**, and Function **200**, as shown in **Fig. 2**, in the present Ontology. These nodes are called Atoms for two different reasons. The first is that they refer to the basic building blocks for any imperative specification. The second reason has to do with the fact that these Nodes serve as terminal nodes for the recursive-traversing interpreter. This point is covered in detail in the description of the recursive-traversing interpreter. In the present implementation, using a single processor computer and a property graph database, each Atom (Data or Function) is represented as a Node with the following properties:

1. Name: (A string of alphanumeric symbols)
2. Value: (A string of alphanumeric symbols or an integer or a real number)
3. ArgValues: (An array of a string of alphanumeric symbols or integers or real numbers)
4. Type: (A string label which is one of the seven types of Nodes)
5. Description/Comments: (Optional)

A unique feature in representing Atoms in this work is the use of the property ArgValues. This is an array, primarily used to keep value inputs from other nodes. The utility of this is explained in the section describing the recursive-traversing interpreter. Another key representational feature,

in the case of Function nodes, is the fact that these nodes too have a Value property. This Value represents the operation/function to be performed by the recursive-traversing interpreter. The name of the Function node does not affect this process and hence a Function node can also be used to represent dynamically evolving processes. Note that only a partial list of properties is shown in both the figures.

It should be noted that in addition to standard functions, a Function node could also have a graph traversal operation as its value, thereby adding to the expressive power of the Ontology.

The second type of nodes described in the present invention is 'Compound'. All Nodes that are not Atoms are called Compounds. Broadly they can be divided into three groups – the first comprising Combiners and Branches, the second comprising Agents and Objects and the final group comprising Models. The division of the Compounds into these three groups is merely for ease in understanding their semantic role and does not play any other role in the Ontology described in the present invention.

Referring to Fig. 3, exemplary embodiment of 'Combiner' is shown according to present invention. The function of the Combiners is to enable the recursive-traversing interpreter (RTI) to evaluate a function by providing it with the function and its arguments. In a more general sense, they provide Objects to Agents to generate output. The input and output Agents, Objects, Functions, Data (or other Combiners or Branches) are indicated by Dataflow edges. (These are one of the three types of edges, the others being Controlflow and Property. These edges are described in detail in a later part of the description.) The RTI computes the value of the

Combiner based on its input (incoming) Dataflow edges and writes the computed value to the Nodes indicated by its output (outgoing) Dataflow edges. Once the output is written, the next node to be traversed by the recursive-traversing interpreter is given by the Controlflow edge leading out from the Combiner Node. As shown in **Fig. 3**, the node **300** is a combiner. The nodes **200** and **100** with names Func (Atom – Function) and x (Atom – Data) respectively and are the inputs to the combiner C1 **300**. The output of the combiner is written to node **100'** named y (Atom – Data). Once the RTI writes the output to Node **100'** y, traversal control is transferred to the unnamed node. It should be noted that edges **304** represent data flow and edge **302** represents control flow.

Referring to **Fig. 4**, exemplary embodiment of 'Branch' is shown according to present invention. Branches too serve the same purpose as Combiners, but with a slight change in their behaviour post output writing. In this case, the next node to be traversed is indicated by one of the multiple outgoing Controlflow edges from the Branch node, with the value computed by the RTI for the Branch node being used to select the appropriate Controlflow edge. These Branch nodes help to create a loop in the present Ontology. Both Combiners and Branches have the same types of properties as described in the previous section. As shown in **Fig. 4**, the node **400**, B1 is a Branch. Its inputs are a logical function L **200** and a data variable x **100**. Its output is written to the data node y **100'**. The logical function **200** can have a value of 0 or 1 in this case. Similar to the combiner, at the branch node **400**, the RTI computes its output and writes it to y. Then, based on L(x) being 0 or 1, it transfers control to node N1 or N2 respectively. It can be seen that in the general case, this can be extended to any number of possible choices beyond 0 and 1.

Referring to **Fig. 5**, exemplary embodiment of ‘Object Node’ is shown according to present invention. Object Nodes are used to group together other nodes. These other nodes could be nodes of any type, including other Object Nodes. The nodes ‘part’ of an Object are linked to an Object Node with the name of the Object using Property Edges. Any other specific property of this relation can be captured in the edge using other appropriate key-value pairs as properties. As explained here, Object Nodes, thus, require only three properties – Name, Type and Description/Comments (which have the same meaning as described in the section on Atoms). As shown in **Fig. 5**, node **500** is an Object. It has two Data nodes **100** named Height and Weight. These two are Atoms. Note that the Object node **500** is like a ‘tag’ that identifies the nodes that ‘belong’ to it. Instead of the Data nodes **100**, it could also have other Object Nodes to form more complex Objects. The Object Node itself does not have any value. The edges **502** in **Fig. 5** are neither Dataflow nor Controlflow edges. Rather they are Property edges.

Referring to **Fig. 6**, exemplary embodiment of ‘Agent’ is shown according to present invention. Agents are similar to Objects but they have some added features. Each Agent **600** has at least one Combiner Node **300**, one Input Node **100** and One Output Node **100**, with the latter two being Nodes of any type. The property edge **602** joining the necessary Combiner Node **300** to the Agent has an added property called ‘Sub-type’ which has the value ‘Initial’. Similarly the edge **604** joining the Input Node has ‘Sub-type’ ‘Input’ and the edge **606** joining the Output Node has ‘Sub-type’ ‘Output’. It is to be noted that there can be multiple Combiner Nodes and Input and Output Nodes linked to an Agent. However, every agent can have only one Node with the edge label initial and this node must be a Combiner Node. If an Agent Node has links to multiple input and/or output nodes, these can be ordered using additional properties on the edges

(Property Edges) linking them to the Agent. Note that these nodes can also be linked (and generally are linked) through Dataflow and/or Controlflow edges to form an imperative sub-graph. Agent Nodes too require the three properties that are needed by Objects with an added property called 'Status'. This property takes on two values (agent-NI and agent-I to specify whether the agent is Not Invoked or Invoked respectively), the use of which is explained in the section describing the recursive-traversing interpreter.

Referring to **Fig. 7**, exemplary embodiment of 'Model Node' is shown according to present invention. A Model Node is very similar to a Combiner or a Branch Node. The only difference is that a Model Node already has its output node value populated. When the recursive-traversing interpreter evaluates a Model Node, it does not compute any function, rather it fetches the value from the output Node indicated by the outgoing Dataflow edge from the Model Node. Thus, a Model Node helps to capture Declarative specifications in this Ontology. It is to be noted that this is significantly different from an RDF like representation as all the necessary imperative steps – functions as well as data are captured as input Dataflow edges to the Model Node. This makes it possible to define new functions (i.e. values inside Function Nodes), which can then be used inside for input to a Combiner or a Branch Node. As shown in **Fig. 7**, node 700 M1 is a Model. It represents the fact that B is the brother of A.

The three types of edges mentioned hereinbefore will now be described in detail. The three types of edges used in the present invention are Dataflow Edges, Controlflow Edges and Property Edges.

Dataflow Edges capture the flow of data required for evaluations in the Ontology. These edges have a property called 'Order' which has an integer value. This value represents the order of this input as required by the Node that this edge points to. Additionally, instead of using the 'Value' property for every node as its value, one could specify the property name to be used as value on the Dataflow edge that links the current node to other nodes.

Controlflow Edges are used to specify the next Node to be traversed by the recursive-traversing interpreter once the current node has been evaluated. They too have the 'Order' property. In case of Branch Nodes being the source of these edges, the value of the Branch Node and the Order of the outgoing Controlflow edge are matched to determine the next Node to be traversed. In case of all other Nodes, multiple outgoing Controlflow nodes may be sorted by Order or their traversal may be parallelized in case the hardware used for the implementation supports parallel execution.

Property Edges are used to form any relation other than those specified by the above two edge types. Primarily they are used to create Objects and Agents by linking an Object or an Agent Node to other nodes. These edges form the residual class of edges and by adding further properties on these edges, all other relationships can be expressed in this Ontology.

As mentioned earlier, the present invention provides an ontology that has imperative as well as declarative characteristics including a recursive-traversing interpreter that enables running imperative queries on the Ontology hereinbefore described. Now, recursive-traversing interpreter is defined herein in detail.

The recursive-traversing interpreter updates the values of various Nodes in the graph as specified in the Ontology model described hereinbefore. In order to do this, it uses its internal library of functions, which in this case is the standard function library provided by most common high-level programming languages. However, this library can be the Instruction Set Architecture of the microprocessor and thus the hardware would be directly linked to the present Ontology. Once a starting node for traversal is specified by the user query, this interpreter does the following steps:

1. If that Node is an Atom, this interpreter updates all other Nodes which are linked from this Node via outgoing Dataflow edges with the value of this Atom.
2. If that Node is a Combiner or a Branch the interpreter then checks if it requires any arguments for its value to be computed. The 'argValues' property of the Node serves to hold this data. If any value is required, then it recursively traverses those nodes, if not, it computes the value of the present node and writes its output to all the Nodes linked from this Node via outgoing Dataflow edges.
3. If the Node is an Object, the interpreter recursively searches the Object by traversing through its Property edges to find the Atom whose value is sought by the Combiner or any other Node that led to the Object being traversed.
4. If the Node is an Agent, the interpreter identifies the Input Nodes and passes on the requirement to the Combiner and then fetches this data (from other input Nodes to the Combiner). During this first call, the Agent Node's status is 'agent-NI', that is, not invoked. Once this call is made and the inputs to the Agent provided, the Agent Node's status is changed to 'agent-I', that is, invoked. This status then enables the RTI to fetch

the value/s from the Output Node/s of the Agent and then update all Nodes linked by outgoing Dataflow edges to the Combiner or Branch that led to the traversal of the Agent Node, with these output values.

5. If the Node is a Model, the interpreter fetches the value/s specified by the Output Node/s (that is, the node/s that is/are connected by outgoing Dataflow edges from the Model Node) and provides this value/s to the Node that led to the traversal of the Model Node.
6. Once the value of any Node is evaluated, the interpreter then traverses the next Node indicated by the outgoing Controlflow edge from the current Node. Only in the case of a Branch Node, the interpreter chooses the appropriate Node to be traversed next based on the value of the Branch Node and the Order of the outgoing Controlflow edges from the Branch Node.

Since this interpreter recursively traverses the graph, it is called a recursive-traversing interpreter. As indicated in point 1 above, the Atoms form the base case of the recursion. In this work, a combination of eager and lazy evaluation is used for this recursive interpreter evaluation. The eager part ensures that when the interpreter computes a Node's value, all the Nodes that receive input from this Node are immediately updated. The lazy part of the interpreter is used in Combiners and Branches. In case of these Nodes, their `argValues` property is erased after every computation. These values are then sought only when the Node is traversed again. This ensures that once a set of values is used for a computation, it is not reused. Also the next fetching of these values is done only when they are required. This strategy plays a key role in enabling looping which simulates recursion in the present Ontology. Another key feature of this interpreter is its ability to reduce Application Programming Interface (API) specification

requirements by enabling a recursive search (exact or approximate search) within objects and agents to find required input nodes. Thus, Objects created by different people can be used by Agents created by different people without each knowing the structure of the other, as long as they use a common name or description keywords to represent the same variable.

Additionally, the RTI implementation has an internal data structure, which is a stack, which it uses to keep track of nodes to be visited. This stack can also be moved to the graph database as a separate node with Property edges leading out from this node indicating the nodes to be visited and the 'Order' property on these edges indicating the order in which the nodes need to be visited by the RTI.

Furthermore, some Data and Function nodes can also be declared to be Non-Atomic so that the behaviour of the RTI when traversing these nodes becomes similar to that when it is traversing a Combiner. This flexibility helps in some specific domains and is more of a convenience rather than a requirement.

Another point to be noted is that multiple Dataflow edges could provide input to the same Node with the same order (that is, as possible candidates for the same value required by the Node). A choice can be made amongst these edges based on any predetermined criterion (say better reliability if the multiple inputs represent data from different types of sensors or lower cost if the multiple inputs are from different Agents, each of which has a different computational cost etc.) or any criterion specified by the user dynamically at run-time.

The ability of an Ontological System to generate new knowledge in response to user queries is a key factor that determines its utility. The system described so far lends itself to any query that requires an evaluation. It is also possible to handle all declarative type queries and inferences in this system that is possible in other RDF like declarative systems. However, an added advantage here, is the ability to compute inverses of (at least some) imperative processes to enable answering a much larger class of user queries, permitting optimization, system design etc. There are three distinct strategies for this automatic inverse computation described as follows:

Structural Inversion: In Structural Inversion, the inversion is done at the Node level with each Combiner or Branch being replaced by a new Combiner or Branch with the Dataflow edges appropriately changed and the Function Nodes replaced with the inverses of the same. If the inverses of the Function Nodes are not known apriori or the sub-graph being traversed is cyclic, then this approach would not be feasible.

In value-based Inversion, there are two approaches possible here:

Numerical: In this a numerical approximation method such as Picard's iteration or Newton-Raphson's method is used, provided the sub-graph being traversed is numerical, acyclic and has all the inputs relevant for the numerical method being used.

Pseudo: Here, we use history of various Agents to group Agents with their inverses for particular sub-domains for these Agents. These are not actual inverses, as they only seem to be inverses for a particular sub-domain. Hence, they are called Pseudo Inverses. It is to be noted that this approach can be adopted only if a history of Agents' action has been recorded and pseudo inverse pairs have been identified prior to evaluating the query.

Probabilistic Inversion: This appears to be the most promising strategy for general cases. This uses Bayes' Theorem to compute inverses. The domains of the input and output variables are discretized (if they are not already sufficiently discrete) and then the conditional probability relation between the new input bins and the output bins are computed. This conditional probability along with a prior distribution for the input variables (estimated or assumed) enables us to compute the inverse by specifying the posterior probabilities of the various output bins using Bayes' rule.

In the present Ontology, machine learning is possible, which is similar to tuning a neural network. Here, learning could also involve changing the values of the Function Nodes in addition to changing the values of the Data Nodes. However, the main strength of neural networks, which is the back propagation algorithm, does not seem to have a ready analogue here. However, this system is suited to another key problem – that of Transferring Learning/Knowledge.

The ability to transfer learning or knowledge from one domain to another is a key challenge in Artificial Intelligence today. One limitation to this is the multitude of data structures used in various domains making it a huge combinatorial exercise in finding all potential isomorphisms between the two systems. It must further be noted here that these isomorphisms need only be approximate in most cases. The present Ontology provides a uniform high-level representation, which thus provides a much reduced search space for these potential isomorphisms. However, the effectiveness and likelihood of finding such isomorphisms in this reduced search space, is yet to be studied.

As described above, an added advantage of the present Ontology is the ability to compute inverses of (at least some) imperative processes to enable answering a much larger class of user queries, permitting optimization, system design etc. Optimization can be of two types in querying the present Ontology. The first type is called a Structural one, where the best Agent to compute a particular output is to be identified. This problem can be solved similar to an edge-weighted shortest path problem. Further, as different Agents can give the same output, but may require different inputs that differentiate them based on their accuracy and precision, the number of inputs needed by the Agent too could add to the cost of the objective function. On the other hand, the accuracy and precision provided by the Agent could then contribute favourably to the objective function.

The second type of optimization involves choosing input values for a Node or a set of Nodes that optimizes the value or value/s for another Node or set of Nodes. This is a standard black-box optimization problem, with the black box being replaced by the sub-graph and thus permitting structural inferences too.

The implementation of the Ontology described hereinbefore uses a property graph database, which is one of the most common types of graph databases. This database can be implemented in a single computer with sufficient hard disk space and for scalability; it can be stored across multiple machines. The storing and data retrieval mechanisms are specific to the graph database implementation and are generally provided by the database provider. The recursive-traversing interpreter can be implemented in any modern computer system with single or multiple

processors. Theoretically, it only requires access to the functions specified as part of the processor architecture. However, for ease of use, the present work uses high-level functions provided by most standard programming languages that then interface with the processor's hardware-software interface. For the purpose of illustration of the system presented, an Apple Macbook Air laptop computer with an Intel 1.86 GHz Core 2 Duo Processor, running Mac OS X 10.7.5 (Lion) was used. The computer had a memory of 2GB 1067 MHz DDR3 and a flash based hard disk of 128 GB capacity. For the graph database, a free open-source graph database was used. The Recursive Traversing interpreter was implemented in Python 2.7.5.

Besides a single computer based operation, the system can be used over a network of computers in two ways. The first approach, which is to handle big data, is to partition the graph over several computers in a network. The use of the present Ontology is to aid the partitioning logic whereby a single Agent or Object is not partitioned into different machines whenever possible. Since the recursive-traversing Interpreter focuses only on local data (neighbouring nodes and edges) for execution, a separate copy of these distributed elements in the graph database could be made in a new machine and post running the interpreter, these (sub-) graphs could be updated in their original locations. This implementation is depicted in **Fig. 8**.

Another approach to scaling this Ontological System is to allow multiple computers (clients) to connect to the computer (or cluster or network) with the graph database (knowledge base server), but each client computer can have its own recursive-traversing Interpreter. The advantage here is that, in this case, each client can have its own implementation of the Atomic Functions and thus

can exhibit different behaviour while accessing the same graph. This architecture is shown in Fig. 9.

Following are the examples carried out on the system and graph database mentioned hereinbefore.

EXAMPLE 1: Euler GCD Computation with Object as Output

Referring to Fig. 10, in this example, the Agent 1002 is used to compute the GCD of the two input Nodes to the Combiner Node 1004. There are 23 nodes in this example numbered from 0 to 22. There are 42 relationships numbered from 0 to 41. The lists of Nodes and Edges in this example are given in Tables 1 and 2 below. The main Combiner in this application is the node numbered 13. The Recursive Traversing Interpreter (RTI) starts from this node. In order to understand the computation, we track the internal stack of nodes to be visited by the RTI and also the argValues and value property of each node that the RTI visits. Initially the values of all Combiners are set to “ ” (blank) and the argValues array is set to “NA” for each element, with the number of elements in the array determined by the number of incoming data edges to the Combiner with unique order property. All the Data and Function Nodes are set to initial values as given in Table 3.

Table 1: List of Nodes

Node Id	Name	Type
0	"if-branch"	"branch"
1	"if function"	"function"
2	"equals"	"data"
3	"zero"	"data"
4	"mod-combiner"	"combiner"
5	"input-m"	"data"
6	"input-n"	"data"

7	"mod-function"	"function"
8	"set function"	"function"
9	"set-combiner-1"	"combiner"
10	"set-combiner-2"	"combiner"
11	"set-combiner-3"	"combiner"
12	"output"	"data"
13	"new-combiner-main"	"combiner"
14	"output object"	"object"
15	"input-n"	"data"
16	"input-Object"	"object"
17	"second object"	"object"
18	"EulerAgent"	"agent"
19	"input-m"	"data"
20	"equals"	"data"
21	"small output object"	"object"
22	"output"	"data"

Table 2: List of Edges

Edge_Id	Type
0	"DATAFLOW"
1	"DATAFLOW"
2	"DATAFLOW"
3	"DATAFLOW"
4	"DATAFLOW"
5	"DATAFLOW"
6	"DATAFLOW"
7	"DATAFLOW"
8	"DATAFLOW"
9	"DATAFLOW"
10	"DATAFLOW"
11	"DATAFLOW"
12	"DATAFLOW"
13	"DATAFLOW"
14	"DATAFLOW"
15	"DATAFLOW"
16	"CONTROLFLOW"
17	"CONTROLFLOW"
18	"CONTROLFLOW"
19	"CONTROLFLOW"
20	"DATAFLOW"
21	"DATAFLOW"
22	"DATAFLOW"
23	"DATAFLOW"

24	"PROPERTY"
25	"PROPERTY"
26	"PROPERTY"
27	"PROPERTY"
28	"PROPERTY"
29	"PROPERTY"
30	"PROPERTY"
31	"PROPERTY"
32	"PROPERTY"
33	"PROPERTY"
34	"PROPERTY"
35	"PROPERTY"
36	"PROPERTY"
37	"PROPERTY"
38	"PROPERTY"
39	"PROPERTY"
40	"PROPERTY"
41	"PROPERTY"

In this example, RTI is asked to start by exploring Node 13. Its internal stack (called fringe) has only Node 13 at this stage. It pops the fringe and traverses to Node 13. It checks the type of the Node and finds that it is a Combiner. It hence checks if all its arguments are present (i.e. there are no “NA” in its argValues array). Initially, it finds that all the required arguments for Node 13 are missing and hence, it adds the required nodes to the fringe. It then proceeds to pop the fringe again and continue the same algorithm. The various steps involved in this are given in the table below. Whenever a node is evaluated, its value is updated to the argValues property of all its outgoing DATAFLOW neighbours with the index of the argValues property array being determined by the ‘Order; property of the DATAFLOW edge.

Table 3

Step No.	Fringe at Start of Step	Current Node	Current Node Type	Current Node ArgValues	Current Node Status	Current Node Value	Fringe at End of Step
1	13	13	Combiner	[NA, NA, NA]	Not Evaluated		13, 15, 16, 18
2	13, 15, 16, 18	18	Agent		Not Evaluated		13, 15, 16
3	13, 15, 16	16	Object		Evaluated	Object Id	13, 15
4	13, 15	15	Data	24	Evaluated	24	13
5	13	13	Combiner	[Agent-Id, Object-Id, 24]	Not Evaluated		13, 18
6	13, 18	18	Agent		Evaluated	Agent Initial Node Id And Input Nodes Data Written (5 and 6 from Object Node 16 and Data Node 15)	13, 0
7	13, 0	0	Combiner	[NA, NA, NA, NA]	Not Evaluated		13, 0, 4, 3, 2, 1
8	13, 0, 4, 3, 2, 1	1	Function	[jklf]	Evaluated	jklf	13, 0, 4, 3, 2
9	13, 0, 4, 3, 2	2	Data	[==]	Evaluated	==	13, 0, 4, 3
10	13, 0, 4, 3	3	Data	[0]	Evaluated	0	13, 0, 4
11	13, 0, 4	4	Combiner	[NA, NA, NA]	Not Evaluated		13, 0, 4, 6, 5, 7
12	13, 0, 4, 6, 5, 7	7	Function	[jkMod]	Evaluated	jkMod	13, 0, 4, 6, 5
13	13, 0, 4, 6, 5	5	Data	[42]	Evaluated	42	13, 0, 4, 6
14	13, 0, 4, 6	6	Data	[24]	Evaluated	24	13, 0, 4
15	13, 0, 4	4	Combiner	[jkMod, 42, 24]	Evaluated	18	13, 0

16	13, 0	0	Combiner	[if == 0 18]	Evaluated, Controlflow edge to 10	False	13, 10
17	13, 10	10	Combiner	[NA, 24]	Not Evaluated		13, 10, 8
18	13, 10, 8	8	Function	[jkSet]	Evaluated	jkSet	13, 10
19	13, 10	10	Combiner	[jkSet, 24]	Evaluated, Controlflow edge to 11	24	13, 11
20	13, 11	11	Combiner	[jkSet, 18]	Evaluated, Controlflow edge to 0	18	13, 0
21	13, 0	0	Combiner	[NA, NA, NA, NA]	Not Evaluated		13, 0, 4, 3, 2, 1
22	13, 0, 4, 3, 2, 1	1	Function	[jkIf]	Evaluated	jkIf	13, 0, 4, 3, 2
23	13, 0, 4, 3, 2	2	Data	[==]	Evaluated	==	13, 0, 4, 3
24	13, 0, 4, 3	3	Data	[0]	Evaluated	0	13, 0, 4
25	13, 0, 4	4	Combiner	[NA, NA, NA]	Not Evaluated		13, 0, 4, 6, 5, 7
26	13, 0, 4, 6, 5, 7	7	Function	[jkMod]	Evaluated	jkMod	13, 0, 4, 6, 5
27	13, 0, 4, 6, 5	5	Data	[24]	Evaluated	24	13, 0, 4, 6
28	13, 0, 4, 6	6	Data	[18]	Evaluated	18	13, 0, 4
29	13, 0, 4	4	Combiner	[jkMod, 24, 18]	Evaluated	6	13, 0
30	13, 0	0	Combiner	[if == 0 6]	Evaluated, Controlflow edge to 10	False	13, 10
31	13, 10	10	Combiner	[NA, 18]	Not Evaluated		13, 10, 8
32	13, 10, 8	8	Function	[jkSet]	Evaluated	jkSet	13, 10
33	13, 10	10	Combiner	[jkSet, 18]	Evaluated, Controlflow edge to 11	18	13, 11
34	13, 11	11	Combiner	[jkSet, 6]	Evaluated, Controlflow edge to 0	6	13, 0
35	13, 0	0	Combiner	[NA, NA, NA, NA]	Not Evaluated		13, 0, 4, 3, 2, 1
36	13, 0, 4, 3, 2, 1	1	Function	[jkIf]	Evaluated	jkIf	13, 0, 4, 3, 2
37	13, 0, 4, 3, 2	2	Data	[==]	Evaluated	==	13, 0, 4, 3

38	13, 0, 4, 3	3	Data	[0]	Evaluated	0	13, 0, 4
39	13, 0, 4	4	Combiner	[NA, NA, NA]	Not Evaluated		13, 0, 4, 6, 5, 7
40	13, 0, 4, 6, 5, 7	7	Function	[jkMod]	Evaluated	jkMod	13, 0, 4, 6, 5
41	13, 0, 4, 6, 5	5	Data	[18]	Evaluated	18	13, 0, 4, 6
42	13, 0, 4, 6	6	Data	[6]	Evaluated	6	13, 0, 4
43	13, 0, 4	4	Combiner	[jkMod 18 6]	Evaluated	0	13, 0
44	13, 0	0	Combiner	[jklf == 0 0]	Evaluated, Controlflow edge to 9	0	13, 9
45	13, 9	9	Combiner	[jkSet 6]	Evaluated	6	13
46	13	13	Combiner	[Agent, Object, 24]	Evaluated, output (marked as output in Agent Node, here Nodes 2 and 12) written to Nodes given by outgoing Dataflow edges	==, 6	

EXAMPLE 2: Food Network

Food networks, which have been described in “Flavor Network and the principles of food pairing, Yong-Yeol Ahn, Sebastian E. Ahnert, James P. Bagrow and Albert-Laszlo Barabasi, Nature, Scientific Reports, Published 15 December 2011” and have been traditionally modelled in a declarative (RDF / first order logic type) form, are modelled in the proposed system. One limitation mentioned in the said reference is addressed using the representation developed in the present invention. This limitation is that of the representation used in said reference not being

able to address the effect of the mode of preparation on the flavour of a dish. Referring to **Fig. 11** it shows two steps in a food recipe given in “<http://www.epicurious.com/articlesguides/bestof/toprecipes/bestpastarecipes/recipes/food/views/Lemon-Gnocchi-with-Spinach-and-Peas-240959>”. In this figure, the representation in the format prescribed in the present work allows one to capture the effect of the mode of the preparation as well as other parameters involved in the preparation. If the Function Atoms are chosen based on what a robotic system can do, then the specification also becomes an executable recipe for that system. Further, this representation also illustrates use of additional properties on edges such as ‘quantity’ as described in the figure.

In the above figure, nodes **200** represent Functions, nodes **100** represent data (food ingredients), nodes **500** represent Objects and nodes **300** represent Combiners (steps in the recipe). The edges **304** represent data flow (flow of ingredients), edges **302** represent flow of control (sequence of steps) and edges **502** represent Property edges which denote membership of an Object and also have a property on them which defines the quantity of each ingredient to be used in the particular step. Using this representation, several additional properties of recipes can be learned. For example, queries such as:

- Which ingredient occurs in maximum number of recipes and is also the single largest component by weight (or cost or volume or importance) of those recipes?
- Which ingredients are used in multiple steps in the same recipe?
- Which ingredients are generally used only in the last few steps of recipes?

- What are the relationships between Functions and Ingredients? Do some Functions always need a particular ingredient (for a trivial example, frying and oil generally occur together) and do some ingredients only pair with specific Functions?
- Is there a relationship between nature of ingredients and number of steps in a recipe?

EXAMPLE 3: Legal Document (Act/Regulations/Agreement etc.)

The statements (clauses) in any legal document can be classified into three types – definitions, facts and action statements. All these three can be represented in the Ontological model described in the present invention. For example, we can consider three typical clauses in a hypothetical legal agreement:

- a. 'Group' means Company A and Company B
- b. The PAT for FY 2013 is INR 1 million
- c. Equity Share Conversion is to be done at a value of 5 times the EBITDA for FY 2014

Referring to **Figs. 12, 13 & 14**, the graphs illustrate the representation of the above three clauses. The advantage of this representation is that one can query about the parameters appearing in various clauses as well as one can compute the effect of execution of the Action Statement clauses. Using inversion and optimization techniques in action statement clauses, simulation of various scenarios as well as identifying the optimum choice of parameters that comply with the specified regulations can be done. Note that Action Statements can also be functions with only side effects, such as 'Filing of a document'. In this case, the output of the combiner would be the

status of completion of the function (i.e. success or failure along with time of execution, acknowledgement etc.)

EXAMPLE 4: Financial Modelling

One of the advantages of using the present work in Financial Modelling is the ability to attach multiple data sources for the same parameter and then choose the required one during query execution. Also, the method of processing various financial parameters could be chosen during query execution based on the accuracy and other requirements of the user. Two examples representing these two types are shown in **Fig. 15** and **Fig.16**.

As shown in **Fig. 15**, Gross Profit is computed as the difference of Revenues and Cost of Goods sold (COGS). But, here, there are two different sources to compute revenue. The multiple revenue options are indicated by the fact that both the data edges have the same 'Order' property. Cost of Equity can be computed using two different methods – Capital Asset Pricing Model (risk free rate + beta * (market return – risk free rate) or can be computed using a (hypothetical) thumb rule such as (risk free rate + inflation rate + 5 %). **Fig. 16** illustrates a graph that permits both these options with two Agents (the nodes belonging to the agents (connected via Property edges) are not shown to reduce clutter). The CAPM agent requires inputs – risk free rate (r_f), market return (r_m) and beta. The ThumbRule Agent requires r_f and inflation rate (r_i). Note that the 'parameters' Object has the inputs required for both these agents. The multiple choices of Agents are indicated by both the data edges having the same 'Order' property.

EXAMPLE 5: Probabilistic Graphical Models

The two main types of probabilistic graphical models are Bayesian Networks and Markov Networks represented by directed graphs and undirected graphs respectively. The theory of Probabilistic Graphical Models demonstrates the construction and equivalence of a graph from a joint probability distribution such that the independence relations specified in the graph are valid in the specification of the joint probability distribution. This graph-based factorization permits one to exploit such independencies and consequently reduce the number of parameters required to state the joint probability distribution. The difference between Bayesian Networks and Markov Networks is that in the former, each factor (that governs dependence of one node on the values of the others) represents a conditional probability distribution (CPD) and thus can be interpreted as possibly representing a causal link. In the latter type, the factors are just functions that assign values to every value combination of the variables in their domain or scope. The concepts of active trails in such graphs help us determine independencies and/or conditional independencies between any sets of variables. In addition to this, local structure of the conditional probability distribution or factor specification permits further reduction in number of required parameters. Referring to **Fig. 17** and **Fig. 18**, a simple Bayesian Network and a simple Markov Network are represented in the Ontology specified in the present invention respectively. In this representation, the concepts of active trails for both the types need to be modified. However, this representation enables representation of multiple CPDs for the same set of variables as can be seen in **Fig. 17** for Bayesian Networks. Here, agents 1 and 2 represent two different CPDs that relate the same three variables A, B and C. Thus, the same graph can be used to capture multiple relationships, which is not possible in the traditional approach. Also, local structure can be easily captured in the Agent definition. In **Fig. 18**, which depicts a Markov

Network, the explicit relationship between factors and variables is captured in the present representation. Again, as in the Bayesian Network case, multiple choices for the factor function between the same set of variables can be represented here. This explicit relationship between factors and variables is not captured in the traditional approach.

Since other modifications and changes to fit particular requirements and environments will be apparent to those skilled in the art, the invention is not considered limited as described by the present preferred embodiments which have been chosen for purposes of disclosure, and covers all changes and modifications which do not constitute departure from the spirit and scope of this invention.

Dated this 8th day of August 2013



(Nikhil Ranjan)
of Khaitan & Co
Agent for the Applicant
Reg No IN/PA-1471