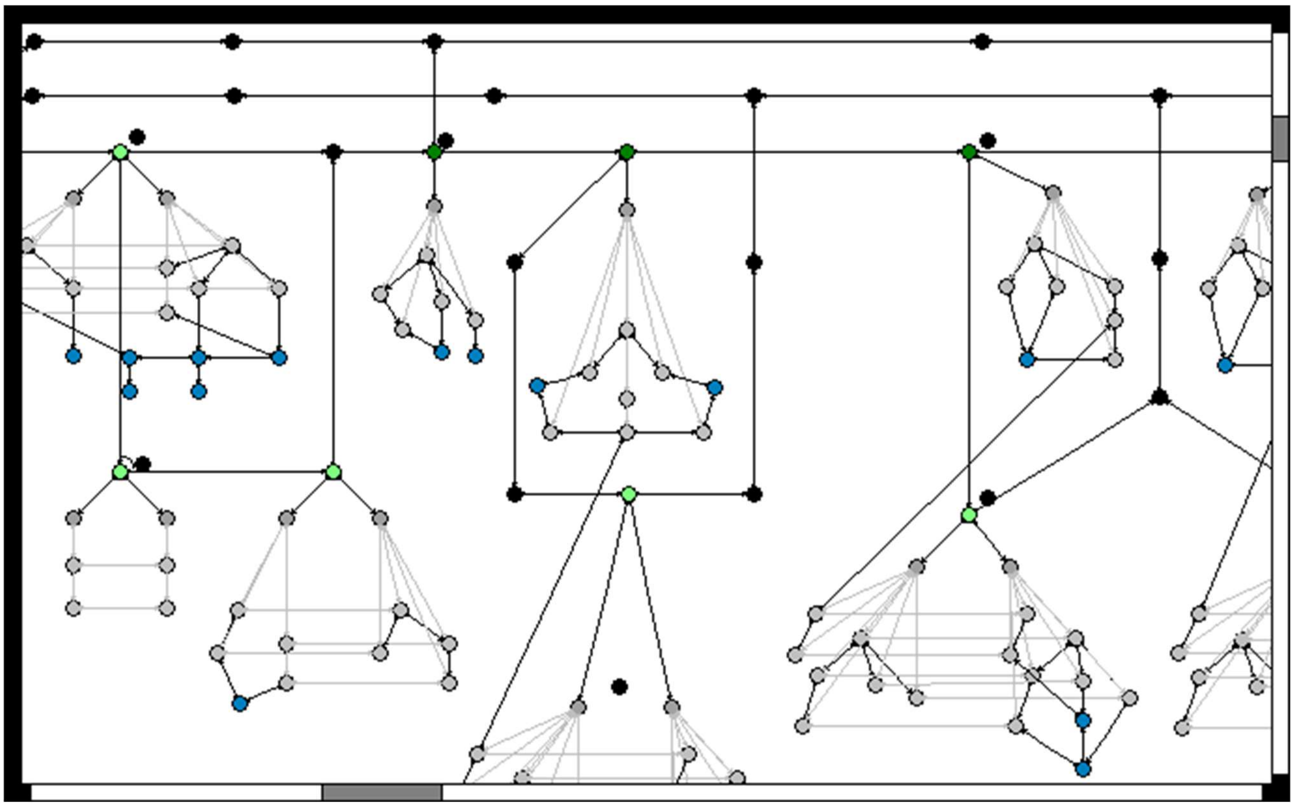
OutOfBrain



Reference Manual
version 1.01

Rockingstone Robotics B.V.
Wageningen, The Netherlands, 2006

Contents

1 Introduction

OutOfBrain is an AI development tool based primarily on the principles of graph rewriting. Unlike most other programming environments, OutOfBrain is not code-oriented. Programming and debugging is done in run-time with OutOfBrain Developer, a graph-oriented graphical user interface which ables the programmer to change the running program without having to shut it down or having to recompile. The client-server architecture allows multiple programmers to work on the same 'source' simultaneously and each programmer can directly follow the work of other programmers. The 'non-code' IDE makes it easy for developers and researchers to communicate their ideas and observations. OutOfBrain is an executable graph combined with the principles of graph-rewriting and autonomous agent technologies. Multiple agents can be created to work together to solve complex problems and achieve goals.

The philosophy behind the use of graphs as opposed to code, is that all data structures can be represented as graphs. In general, code-oriented data structures are limited to records, collections, lists, trees, and combinations of these. Graphs, however, can directly describe all types of structures in a more direct and intuitive way. The developers of OutOfBrain expect the advantages of the use of graphs to become apparent when dealing with complex problems, such as those present in the field of AI.

2 The OutOfBrain architecture

The graph-structures in OutOfBrain are stored in an in-memory graph database. The content of this database is viewable with the OutOfBrain Developer (OOBD). The same content is parsed and processed by the OutOfBrain Virtual Machine (OOBVM), integrated into the OutOfBrain Server. Multiple OOBD clients can connect to the server, allowing multiple programmers to work on the same "source". Interfacing with applications is done via TCP, as TCP servers and clients can be constructed in the form of graphs.

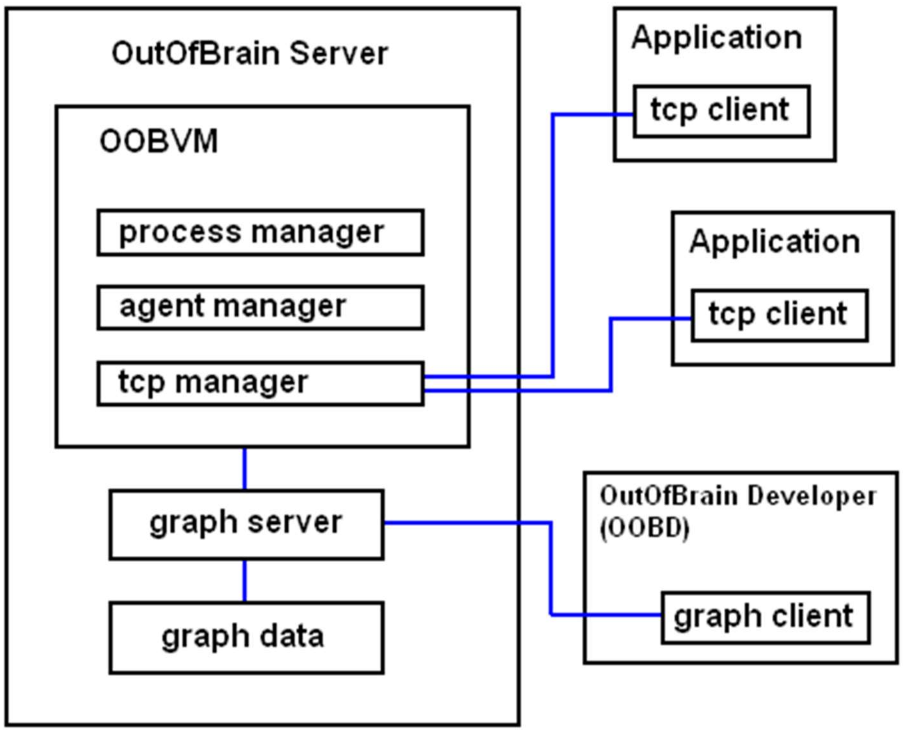


Figure 2.1 *The OutOfBrain architecture*

3 The OutOfBrain Developer (OOBD)

This chapter describes the available commands for opening, saving, creating and navigating graphs in OutOfBrain Developer (OOBD). OOBD is a stand-alone version of OutOfBrain, complete with viewer/editor and integrated OOBVM. There is also a client version called OutOfBrain Developer Client (OOBDC), which can connect to the server version of the OOBVM. In this chapter, however, we just describe the OOBD program.

3.1 Opening the OOBD application

Run the application "outofbrain.exe" to launch OOBD. By default, the program will open the graph file named "main.gd". If no such file exists, an empty graph named "main.gd" is created, but not saved to disk. OOBD is a simple interface with just a single menu called "File". The white background is the graph itself.

3.2 Creating a new graph

To create a new graph, select "New" in the File menu. OOBD will prompt to ask whether the currently opened graph should be saved. Press the appropriate button and a new empty graph will be created called "untitled.gd". At this point the new graph is not yet saved to disk.

3.2 Opening a graph file

To open a graph file, select "Open" in the File menu (or press Ctrl-N). A file-dialog with the caption "Open" will be displayed. Select the desired graph file and press the Open button. Before opening the selected graph file, OOBD will prompt to ask whether the currently opened graph should be saved.

3.3 Saving a graph to file

To save graph to disk, select "Save" in the File menu (or press Ctrl-S). If the graph is titled "untitled.gd", then OOBD will display a "Save as" dialog. To save the graph under a different file name than it currently has, select "Save As..." in the File menu. OOBD will display a "Save as" dialog.

3.4 Exiting OOBD

To exit OOBD, select "Save and exit" in the File menu (or press Ctrl-E). OOBD will save the changes to the graph file and exit without prompting. To exit OOBD without saving changes to the graph file, select "Exit without saving" in the File menu (or press Ctrl-W). Another method to close OOBD is to press the Close in the system menu or by pressing the close-cross button in the upper right corner. This way of closing will save the changes automatically before shutting down.

3.5 Scrolling though a graph

If a graph does not fit within the screen window, OOBD will show scrollbars on the bottom and right sides of the window. These scrollbars will adjust automatically when the lowest vertex or the far-right vertex is moved. To move the window over another part of the graph, click and drag the appropriate scrollbars. The mouse-wheel operates the vertical scrollbar. The mouse-wheel operates the horizontal scrollbar while holding down the Ctrl-key. To move the window over the graph using the keyboard, use the arrow keys. To move the window up or down over the graph, use the Page up and Page down keys, respectively. The Home key positions the window at the top of the graph. To position the window at the top-left-hand corner of the graph, press Ctrl-Home. The End key positions the window at the bottom of the graph.

3.6 Zooming out

Graphs can become quite large, so it is sometimes difficult to navigate to the desired section. In this situation the zoom-function can be helpful. To zoom out, press the Z-key on the keyboard. The OOBD will then show the graph with a zoom-out-factor of 4. At this scale, the labels are not shown and cannot be edited. The border of the original view-window is displayed as a rectangle. To return to the original unzoomed view-window, press the R-key on the keyboard. When moving the mouse over the zoomed-out graph, a similar rectangle is displayed around the mouse cursor. To zoom in onto this rectangle, either press the right mouse button or press the Z-key. To zoom out temporarily, hold down the right mouse button. The view-window rectangle can then be moved to another area, and will become the new view window when releasing the right mouse button. When in zoom-out mode, you can center the view window rectangle by pressing the M-key on the keyboard.

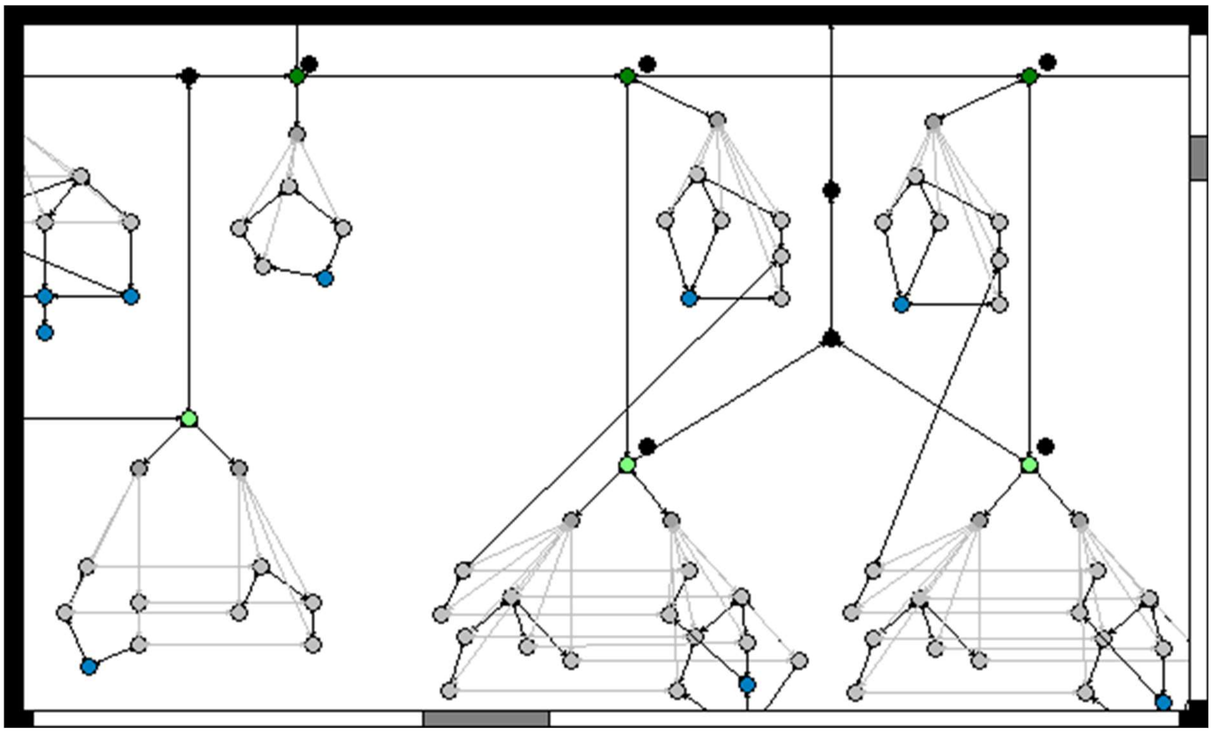


Figure 3.1 *Zooming out.*

4 Editing graphs in OutOfBrain Developer

This chapter describes the available commands for altering graphs. Whenever the text uses the terms "click" or "mouse-button", the left-mouse button is implied. Whenever the text uses the term "key", a button on the keyboard is implied. The chosen terminology for graph components are "vertices" and "edges". In OutOfBrain edges are directional, directed from the "from-vertex" and directed to the "to-vertex". Multiple edges between vertices are supported. Edges with the same from-vertex and to-vertex are also supported. "Dangling edges" are not supported. Vertices and edges each have a one-line textual label. An edge with from-vertex A and to-vertex B is said to be A's "from-egde" and B's "to-edge".

4.1 Adding a vertex

To add a vertex, place the mouse cursor at the position where the vertex is to be placed and press the Insert key. A new vertex will be created with an empty label.

4.2 Deleting a vertex

To delete a vertex, first click the appropriate vertex. Be sure to click the center part of the vertex, and not the black border. The vertex now has a red border showing that it is selected. To delete the selected vertex, click the Delete key. The Delete key deletes all selected vertices. Deleting a vertex will automatically delete all connected edges.

4.3 Moving a vertex

To move a vertex, click and drag the appropriate vertext with the mouse. Be sure to click on the center part of the vertex, and not the black border.

4.4 Labeling a vertex

Each vertex has a label. To assign or change the label-text, click the label with the mouse. A blinking caret will display, allowing text to be entered with the keyboard. Once edited, press the Enter key to save the entered text. Alternatively, click the mouse outside the label to save the text. To cancel the editing, press the Escape key to restore the original label value. When the caret shows, Ctrl-C will copy the text onto the clipboard. Ctrl-C will replace the label-text with that on the clipboard.

4.5 Selecting and deselecting multiple vertices

To select a single vertex, simply click the vertex. Be sure to click on the center part of the vertex, and not the black border. Selecting a vertex in this way will deselect any other selected vertices. To select more vertices, hold the Ctrl key while clicking vertices one-by-one. To deselect a vertex, hold the Ctrl key while clicking the selected vertex. To deselect all vertices, click anywhere in an empty (white) area.
Another method for selecting a group of vertices that are positioned close to each other, is to position the mouse cursor over a white area, hold down the mouse-button, and drag the red box that appears over the group of vertices. Once all desired vertices are selected, release the mouse-button. This also works in zoom-out mode.

4.6 Moving multiple vertices

To move a group of selected vertices, click and drag any one of the selected vertices, drag the group to the desired position and release the mouse-button. This also works in zoom-out mode.

4.7 Copying and pasting

To copy a selected group of vertices, press Ctrl-C. This will place all selected vertices and interconnecting edges onto the clipboard. To paste the group of vertices and edges, scroll to the desired area and press Ctrl-V. Upon doing so, the original group of vertices will become deselected, and the newly placed group will be selected.

4.8 Adding an edge

To connect an edge from one vertex to another, click the black border of the from-vertext. While holding down the mouse-button, drag the edge to the to-vertex. Release the mouse-button and a new vertex with an empty label will be created. It is possible to add multiple edges between two vertices, and to add edges from and to the same vertex.

4.9 Deleting an edge

To delete an edge, first click the appropriate edge on either the arrow head, the arrow tail, or any of the two circles at the center. The edge is now colored red showing that it is selected. To delete the selected edge, click the Delete key.

4.10 Moving an edge

To change the edge's from-vertex to another vertex, click and drag the circle on the edge that is closest to the from-vertex. Drag the edge to the new from-vertex and release the mouse-button. Changing the to-vertex is analogues to changing the from-vertex.

4.11 Naming an edge

Each edge has a label. To assign or change the labe-text, click the label with the mouse. A blinking caret will display, allowing text to be entered with the keyboard. Once edited, press the Enter key to save the entered text. Alternatively, click the mouse outside the label to save the text. To cancel the editing, press the Escape button to restore the original label value. When the caret shows, Ctrl-C will copy the text onto the clipboard. Ctrl-C will replace the label-text with that on the clipboard.

4.12 Navigating across edges

When the from-vertex and the to-vertex of an edge is placed far away from each other in such a way that both do not fit within the window, then it can be difficult to follow the edge from one vertex to the other vertex. In such a case, click the arrow-head or arrow-tail to move the window over to the other vertex.

5 The OutOfBrain Virtual Machine (OOBVM)

The OutOfBrain Virtual Machine (OOBVM) is a background process which interprets and executes the graph. There are a number of graph-structures that are recognized by the OOBVM. These structures form the "grammar" or the "instruction-set" of OOBVM. Programming in OutOfBrain is therefore completely graph-oriented. The graph structures which are part of the instructions are collectively defined as the "instructions graph". The graph structures which are not part of the instructions are collectively defined as the "application graph". The instructions graph and the application graph both reside in the same graph. Edges which connect between instruction and application are called "interface edges".
Most label-texts used in the instruction set are enclosed by "<" and ">" characters (e.g. "<process>"). These label-texts are recognised by the OOBVM, so these are to be treated as reserved words. The programmer should therefore avoid using label-texts in the application graph that have either the "<" or the ">" characters. This chapter will describe the basic grammar of OutOfBrain.

5.1 <group>

The group structure is frequently used in OutOfBrain. It is essentially an isolation of a sub-graph. The group structure is mostly used as part of an instruction. A group is represented as a vertex labeled "<group>", having from-edges labeled "<group>" pointing to a number of vertices. These vertices are "members" of the group. See an example of a group structure in figure 5.1.

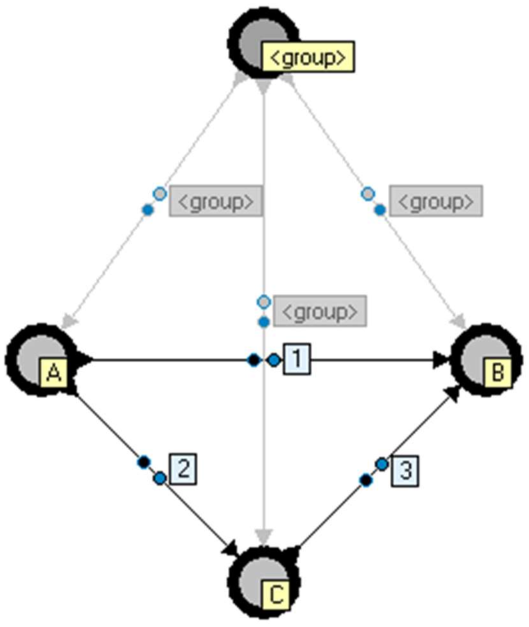


Figure 5.1 *The <group> structure*

All edges that exist between any of the group vertices are also considered members of the group. It is important to note that the members of any group are part of OutOfBrain's "instruction graph". Group structures are used a lot in OutOfBrain, and they can be cumbersome to construct manually. The OOBD supports a keyboard shortcut for constructing a group structure. Just select all vertices which are to be members of the group and press Ctrl-G.

5.2 <process>

Any vertex labeled "<process>" (unless it is a member of a <group>) represents a running process. Many processes can be added to an OutOfBrain application. The main loop running in OOBVM continuously searches for vertices which are labeled "<process>". For each <process>, the OOBVM determines whether it has a leaving-edge labeled "<run>". If such a connection exists, the corresponding to-vertex is "executed" as an instruction. After execution executed, the result of the instruction performed determines where the <run> connection should be moved to. All instructions have a result. For example, if the result of the instruction is "<true>", then the <run> connection is moved to the next instruction pointed at by "<true>". If the instruction is not recognised by the OOBVM, then the (default) result is "<next>". This is shown as an example in figure 5.2. If there is no edge found that corresponds to the result of te instruction, then the OOBVM determines whether there is an edge labeled "<next>". If this is the case, then this edge is followed instead.

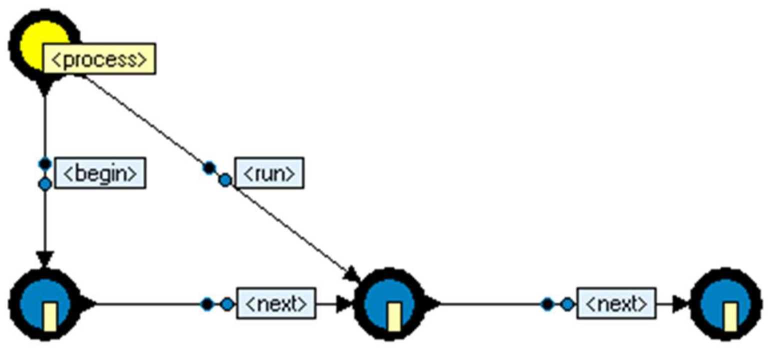


Figure 5.2 *The <process> structure executing an instruction*

If a <process> vertex does not have an edge labeled "<run>", OOBVM determines whether there is an edge labeled "<begin>". If such an edge exists, an edge labeled "<run>" is created with the same from-vertex and to-vertex as the "<begin>" edge. OOBVM then processes the "<run>" edge as previously described.
If the <process> does not have an edge labeled either "<begin>" or "<run>", then the process is "terminated" or "idle".
If there is no from-edge that matches the result of an instruction, then the <run> edge is moved back to the <begin>'s to-vertex. If there is no <begin> edge, then the <run> edge is destroyed and the process goes idle.
A process with a from-edge labeled "<begin>" is therefore a continuous running loop.

5.3 <branch>

The <branch> structure is similar to a "program block" in 'normal' programming. It has a leaving-edge labeled "<begin>" pointing to the <branch>'s first instruction to be executed. Initially an edge labeled "<run>" is created with the same from-vertex and to-vertex as the "<begin>" edge. The instruction pointed to by the <run> edge is executed, and the <run> edge is moved to the next instruction accordingly. The <branch> therefore acts as a <process>, except that it only runs a "single loop" when activated by a <process>. During execution of the <branch> structure, the <process> keeps its <run> edge pointing to the <branch> vertex. Once the <branch>'s <run> edge loops back to its <begin> instruction, then the <process> advances its own <run> edge to the next instruction pointed at by the edge labeled "<next>". The result of a <branch> is therefore always "<next>".

5.4 <execute>

An <execute> instruction is a reference to another instruction. The actual instruction to be executed is the to-vertex of the <execute>'s from-edge labeled "<run>". The <execute> instruction inherits the result of the actual instruction being executed.

5.5 <debug_pause> and <debug_step>

Any vertex labeled "<debug_pause>" (unless it is part of a <group>) will cause the OOBVM to pause. All <process> structures will halt execution until there is no vertex labeled "<debug_pause>". A vertex labeled "<debug_step>" has the same effect, except that the programmer can use the F8 key in OOBD to "step" the processes. Upon each press of the F8 key, all processes execute just one instruction (or branched instruction).

5.6 <implication>

The <implication> is the most-used instruction in OutOfBrain. It is comparable to a single-pushout "rewrite rule" in graph rewriting. It has a "left side" called "premise" and a "right side" called "conclusion". When an <implication> is executed, the OOBVM searches the application graph for patterns that match the premise. The pattern that is found is called the "redex". The labels of the edges and vertices of the redex must match the labels of the edges and vertices of the premise group. Labels of premise vertices and premise edges that are empty are treated as "wild-cards". When a redex is found, the redex is manipulated to complete the pattern represented in the conclusion. The <implication> can be thought of as a "search and replace" function. On another level of abstraction it can be thought of as an instance of causality.
The <implication>'s left and right side are represented as <group> structures. The <implication> has a from-edge labeled "<premise>", and a from-edge labeled "<conclusion>", respectively. Each edge points to a <group> structure. The vertices that are part of the premise group are called "premise vertices", and vertices that are part of the conclusion group are called "conclusion vertices". Premise vertices may have from-edges labeled "<transition>" that connect to conclusion vertices. These edges indicate how the premise vertices map to the conclusion vertices.
When an <implication> instruction is executed, the OOBVM searches the application graph for sub-graphs that match the premise graph, with the additional condition that the redex must not resemble the conclusion. In other words, "find an instance of the premise which is not yet like the conclusion". If such a redex is found, then the implication is said to be in the "before-state". Else, if a redex was found that matched the conclusion but not the premise, then the implication is in the "after-state". If a redex was found that matched both the premise and the conclusion, then the implication is in the "both-state". If a redex could neither be found for the premise or the conclusion,

then the implication is in the "neither-state". The before situation has therefore the highest priority in OOBVM's search algorithm. If multiple redexes are found, then one is chosen at 'random'.
If the implication is in the before-state, then the OOBVM manipulates the redex so that it matches the conclusion. This phaze is called the "imply-state". These manipulations may include creating vertices, labeling vertices, destroying vertices, creating edges, moving edges, labeling edges and destroying edges. After applying these changes, the implication goes into the "implied-state".

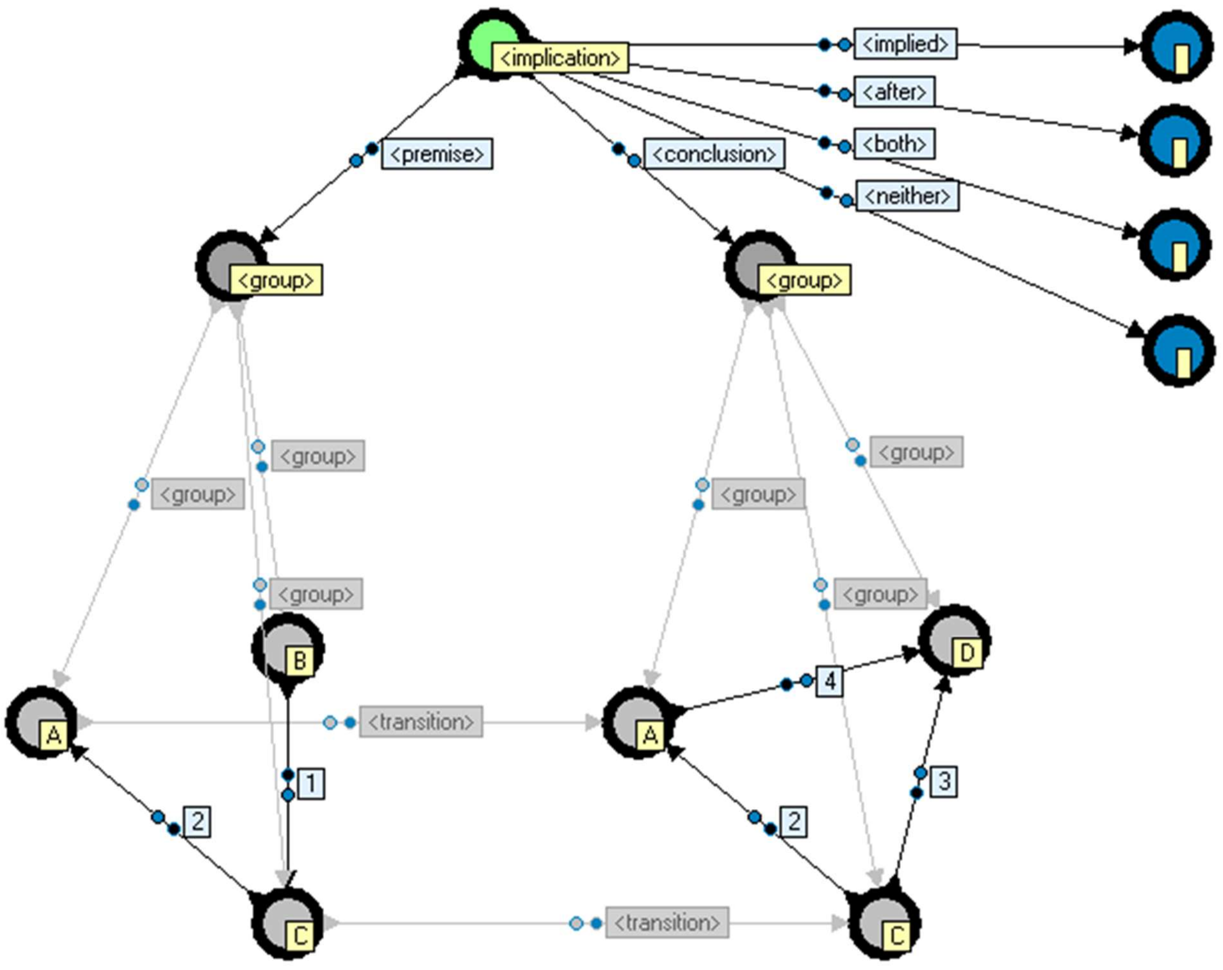


Figure 5.3 *The <implication> structure*

The result of the <implication> is "<implied>" for the implied-state, "<after>" for the after-state, "<both>" for the both-state, and "<neither>" for the neither-state. The result tells the <process> which path to follow to determine the next instruction to execute. Optionally, an edge labeled "<result>" can be connected from the <implication> to another vertex. The result of the implication will be stored in the vertex's label after execution. This is useful during debugging.
In the example shown in figure 5.3, the implication in a before-state would alter the found redex by deleting the vertex labeled "B", delete the edge labeled "1", add a vertex labeled "D", add an edge labeled "4" between vertices "A' and "D", and add an edge labeled "3" between vertices "C" and "D". Important to note is that the labels of the vertices and edges in the redex

must match those in the premise part. If a label in the premise part is en empty string, then this is treated as a 'wild-card'. In such a case, the corresponding redex vertex or edge may have any label.
Implication structures are used a lot in OutOfBrain, and they can be cumbersome to construct manually. The OOBD supports a keyboard shortcut for constructing an implication structure. Just construct a premise situation, select all the vertices and press Ctrl-I. And finally, alter the <conclusion> structure to achieve the desired rewrite rule.

5.7 <condition>

The <condition> instruction is very similar to the <implication> instruction. Except for the instruction's label, the remaining structure is identical to that of an <implication> structure. However, when a <condition> instruction is executed, it does not apply the rewrite rule when it results in the before-state. Instead, it remains in the before-state and returns "<before>" as a result. The <condition> instruction is therefore purely conditional.

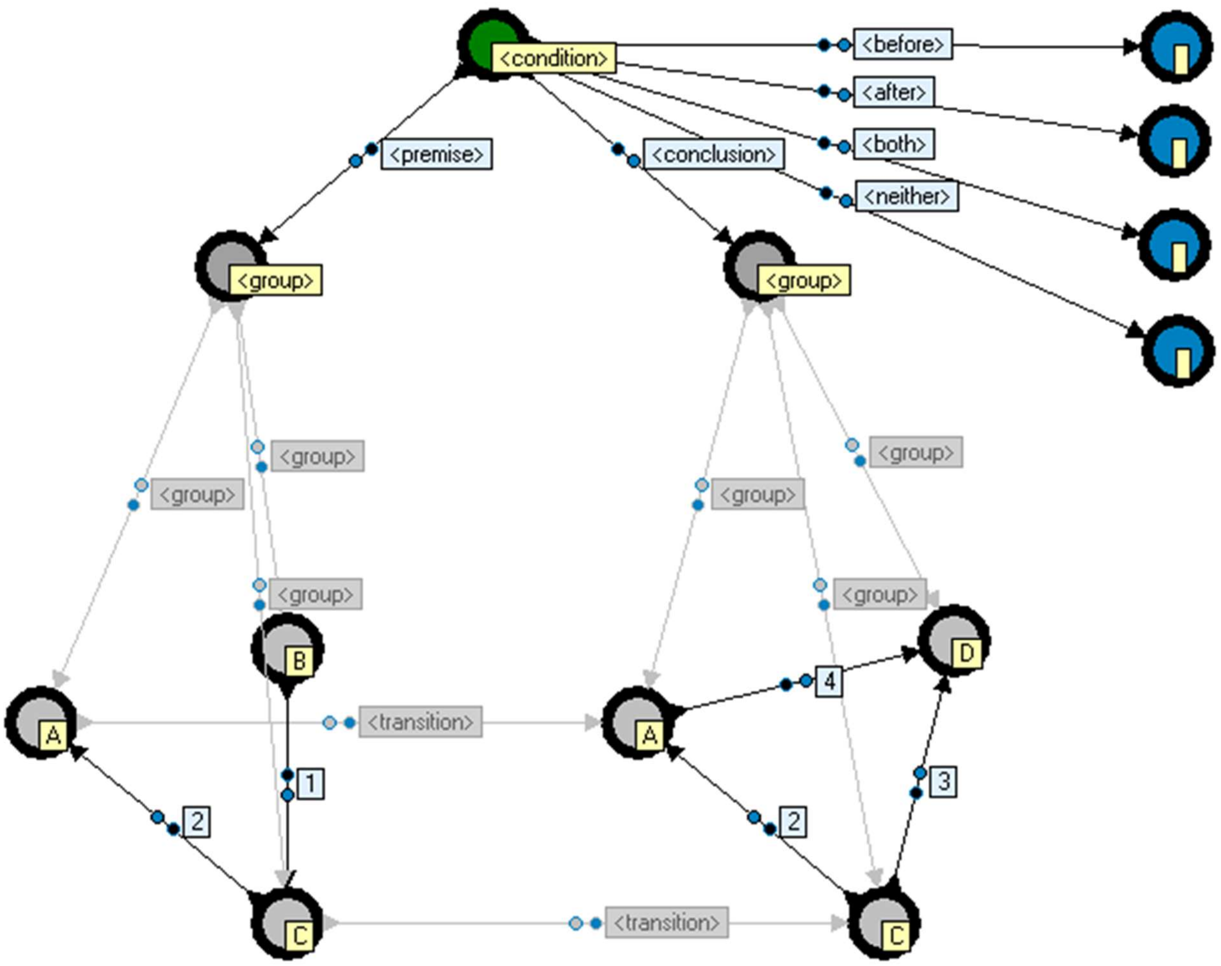


Figure 5.4 *The <condition> structure*

The <conclusion> group of a <condition> structure is optional. If the conclusion group is ommited, then the OOBVM simply determines whether a redex can be found that matches the premise group. If so, then the result is "<true>". Else the result is "<false>". A <condition> structure that has both a <premise> and a <conclusion> group is called a "double condition". A <condition> structure that has just a <premise> group is called a "single condition".

5.8 <wait>

The <wait> instruction is very similar to the <condition> instruction. Except for the instruction's label, the remaining structure is identical to that of a <condition> structure. However, when a <wait> instruction is executed, the <run> edge is not moved on to the next instruction until the condition returns either <after> or <both>. The result of the <wait> instruction is then <next>.

5.9 <candidate> edges

When the OOBVM searches the application graph for patterns that match the premise part of either an <implication> or a <conclusion>, it does so in the entire application graph and may find multiple redexes. In some cases we want to implicitly tell the OOBVM that a particular premise vertex corresponds with a particular application graph vertex. To do so, connect an edge from the premise vertex to the application graph vertex and label it "<candidate>". This forces the OOBVM to use this "candidate vertex" in the redex, if such a redex exists. See figure 5.5 for an example.

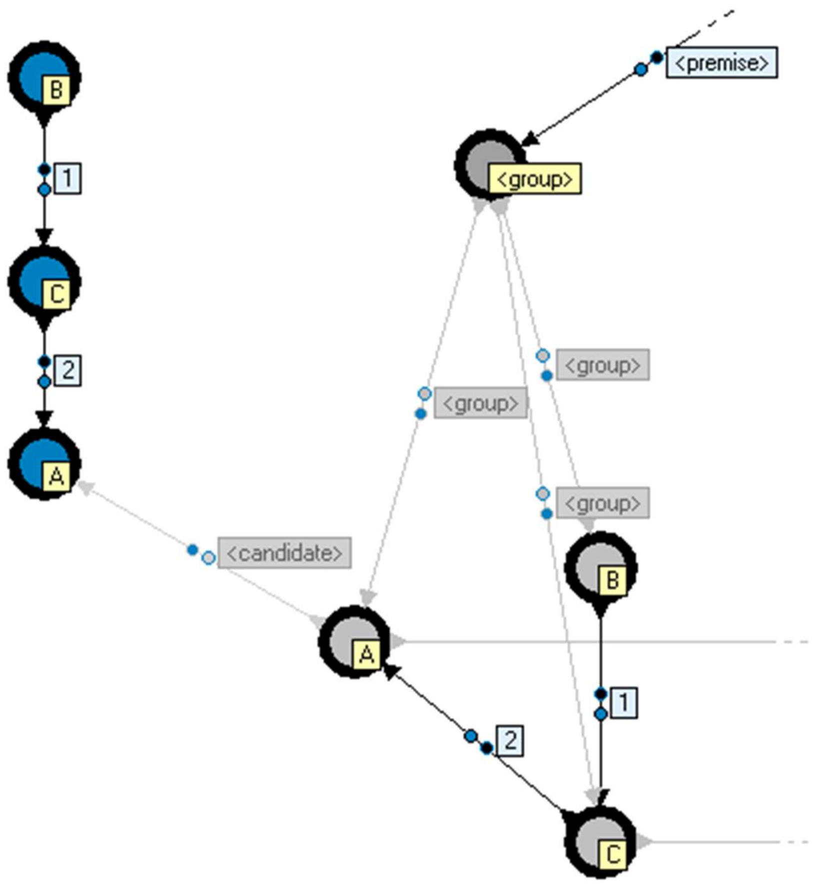


Figure 5.5 *The <candidate> edge*

Multiple <candidate> edges can be added. If a single premise vertex has multiple candidates, then the OOBVM searches for a redex where at least one of these candidates produce a match.

5.10 <reference> and <match> edges

When the OOBVM finds a redex for either an <implication> instruction or a <condition> instruction, then we might want to reference (parts of) this redex in a later instruction. This is especially the case where multiple redexes may be present in the application graph, and we want to focus on a particular redex for successive instructions . A <candidate> edge will not suffice here because we do not know beforehand which redex is selected. For such purposes the OOBVM supports the use of edges labeled "<reference>". The <reference> edge is very similar to the <candidate> edge, except that the <reference> edge points to a premise vertex of another instruction. Consider an instruction A that is followed by an instruction B. Further assume that there exists an edge labeled "<reference>" that points from a premise vertex PB in instruction B to a premise vertex PA in instruction A. When the OOBVM executes instruction A and finds a redex, an edge labeled "<match>" is created between vertex PB and its corresponding redex vertex RA. When instruction B is executed, then vertex RA functions as a candidate for finding PB.

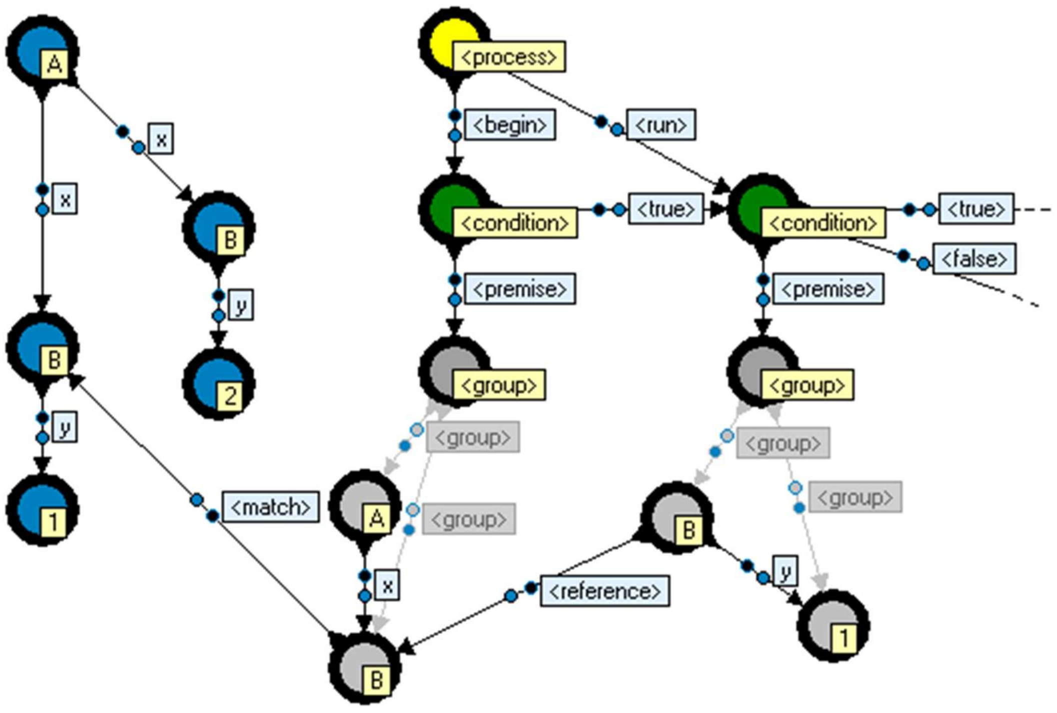


Figure 5.6 *The <reference> edge and <match> edge*

Figure 5.6 illustrates the use of the <reference> edge. The first instruction finds one of multiple possible redexes. The OOBVM detects the presence of a <reference> edge and creates a <match> edge to indicate the corresponding redex vertex. The second instruction then uses this redex vertex as the candidate.

5.11 Conditional operators

The OOBVM supports additional conditional operators that can be attached to the premise groups of instructions. These operators are used by the OOBVM when when it searches for a redex. A redex must meet the conditions specified by these operators. There are five different conditional operators, shown in table 5.1.

conditional operator	description
=	equals
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Table 5.1 *The conditional operators*

To illustrate the use of a conditional operator, consider an instruction A with a premise group P where vertex PA is one of the premise vertices of P. An edge labeled "=" is connected from vertex PA to another vertex V that is not a member of P. Vertex PA has an empty label and V is labeled "5". The edge labeled "=" is a conditional operator. This conditional operator tells OOBVM that the redex vertex of PA must be labeled "5". For an edge to be interpreted as a conditional operator, the following conditions must be met:

1. The edge must be labeled either "=", ">", "<", ">=" or "<=".
2. The edge must be directed from a premise vertex that has an empty label.
3. The edge must be directed to a vertex outside the premise group.

The conditional operator points to a vertex whose label represents the conditional value. The label is interpreted either as a number, as a string or as empty. If the label is interpretable as a number, then the operator is applied as a numeric condition. If the label is not a number and is not empty, then the label is interpreted as a string and the operator is applied as a string condition. For numeric conditions the operators are self-explanatory. The compare operation for string conditions is based on the 8-bit ordinal value of each character.
If the label is empty then the conditional value is a calculated value. Calculated values are determined by the value operators. There are five different value operators, shown in table 5.2

value operator	description
=	is equal to
+	add
-	subtract
*	multiply
/	divide

Table 5.2 *The value operators*

A value operator is an edge that is labeled with one of the value operators listed in table 5.2. The edge can either point to a vertex within the premise group, or to another vertex outside the premise group.

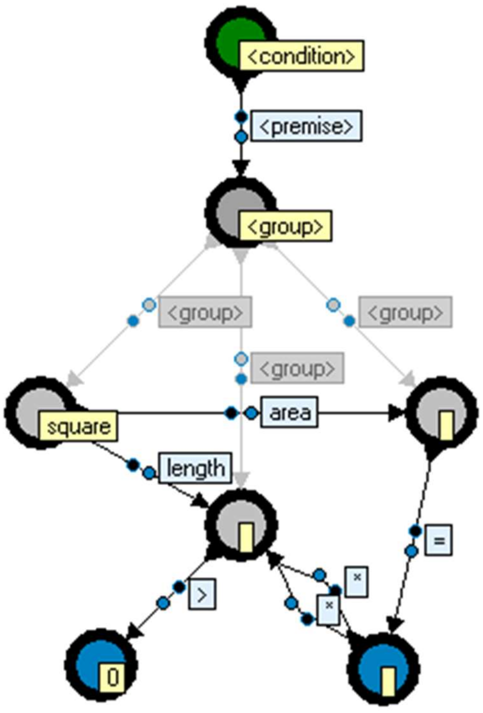


Figure 5.7 *Conditional operators and value operators.*

Figure 5.7 illustrates the use of conditional operators and value operators. The edge labeled ">" tells the OOBVM that the label of the redex vertex must numerically be interpretable as greater than zero. The edge labeled "=" indicates that "the area must equal (length * length)". Multiple value operators can be used to determine a conditional value. The following formula is applied when multiple value operators are used:

conditional value = (((sum("+") - (sum("-")) * (product("*"))) / product("/")

If any of the operator-types are not present, then they are eliminated from this expression. In general, value operators are only used for numeric calculations, and not applied as string operators.

5.12 Assignment operators

Asignment operators are very similar to conditional operators. Conditional operators are attached to premise vertices, but assignment operators are attached to conclusion vertices. The assigment operators are only used in conjunction with <implication> instructions. The assignment values are used to assign values to the labels of redex vertices when an <implication> instruction reaches the imply-state. There are five different assignment operators, shown in table 5.3.

assigment operator	description
=	set to
>	set greater than
<	set less than
>=	set greater than or equal to
<=	set less than or equal to

Table 5.3 *The assigment operators.*

To illustrate the use of an assignment operator, consider an <implication> instruction A with a conclusion group C where vertex CA is one of the conclusion vertices of C. An edge labeled "=" is connected from vertex CA to another vertex V that is not a member of C. Vertex CA has an empty label and V is labeled "5". The edge labeled "=" is an assignment operator. This assignment operator instructs OOBVM to set the label of the redex vertex that corresponds to CA to "5" during the imply-state of A. For an edge to be interpreted as an assignment operator, the following conditions must be met:

1. The edge must be labeled either "=", ">", "<", ">=" or "<=".
2. The edge must be directed from a conclusion vertex that has an empty label.
3. The edge must be directed to a vertex outside the conlcusion group.

The assignment operator points to a vertex whose label represents the assignment value. The label is interpreted either as a number, as a string or as empty. If the label is interpretable as a number, then the operator is applied as a numeric assignment. If the label is not a number and is not empty, then the label is interpreted as a string and the operator is applied as a string assignment. For numeric assignments, the assignment value is set in such a way that the assignment operator would pass as a conditional operator. For example, when a redex vertex is labeled "4", but by means of assignment operators is required to be '>= 5', then the label of the redex is set to 5. But if in this case the redex were labeled "6", then the label would remain "6". For string assignments only the "=" operator is supported.

If the label is empty then the assignment value is a calculated value. These calculated values are determined in the same way as the calculated values are calculated for conditional operators, which is explained in the previous chapter.

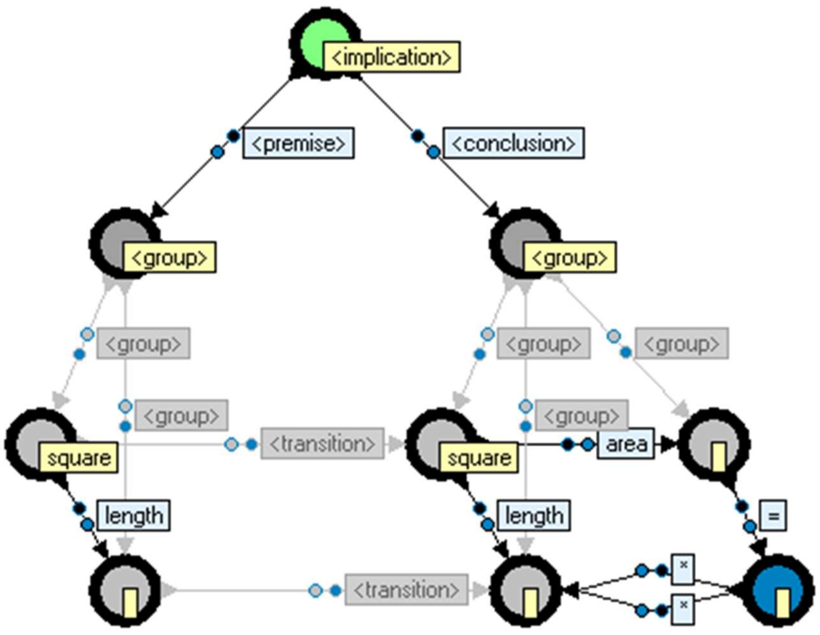


Figure 5.8 *The assignment operator.*

Figure 5.8 shows an example of an assignment operator. The assignment operator is the edge labeled "=". When the implication instruction reaches the imply-state, a new vertex is created and a new edge is created. The newly created vertex is labeled with the assignment value. The assignment value in this case is the "square of the length".

6 OutOfBrain Agents

OutOfBrain has built-in functionality for simulating autonomous agents and multi-agent systems. The type of agents that are modelled in OutOfBrain may not comply with the standard types of agents, and do not fit many of the very many different definitions of an agent. Instead, the agents in OutOfBrain are "graph-rewriting agents". These agents are run by the "Agent Manager" (AM) which is embedded within the OOBVM.

6.1 The Agent Manager

The Agent Manager (AM) is run within the OOBVM. The AM is activated when a <process> runs a vertex that is labeled "<agent>". The <agent> is parsed and executed by the AM. Each agent has a set of connected <implication> and <condition> structures which are loaded into the AM. These implications and conditions are called "branch-implications" and "evaluator-conditions", respectively.
The main component of the AM is the "Virtual Context Generator" (VCG). The VCG basically builds many copies of the entire graph and performs operations on each. Each of these graphs is called a "virtual context", which one can interpret as a "possible world". When an agent is run, the AM determines all the possible results of executing each branch-implication. There may be many different outcomes because every implication may find more than one redex. Each result is stored as a virtual context. This process is repeated for each virtual context recursively, resulting in a tree of virtual contexts constructed in breadth-first fashion. This tree is called the "Virtual Context Tree" (VCT). For every virtual context, the set of evaluator-conditions are evaluated. The results of these conditions tell the AM how "desirable" the virtual context, how "possible" the virtual context is, and whether the virtual context is permitted. Some branches may terminate, either because no branch-implications can be applied, or because a branch-implication may result in a virtual-context which is not permitted. When all virtual contexts are determined, or after a set time period, the AM determines the most desirable virtual context and records the sequence of branch-implications that were necessary to reach that virtual context. This sequence is called the "inference", and forms the "plan" of the <agent>. The inference is then stored as a sequence of instructions that are connected to <agent> vertex. The <process> structure can then execute this inference with the intention to transform the graph to match the desirable virtual context.

An <agent> is connected by edges to several <group> structures. The labels of these edges tell the agent what types of <group> structures they represent.

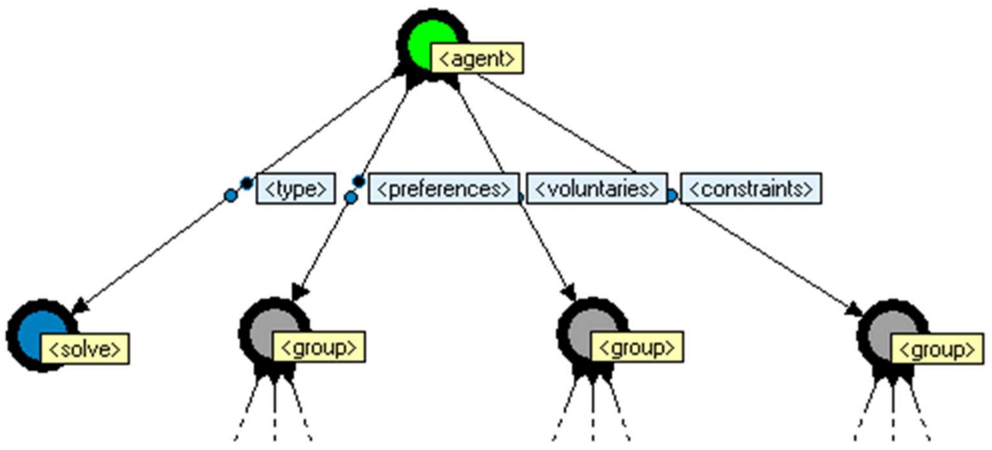


Figure 6.1 *The <agent> structure.*

The meaning of these different types of groups is explained in further detail in the following chapters.

6.2 <voluntaries>

The <voluntaries>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts in the VCT, and are classified as branch-implications. A voluntary implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the voluntary implication is ignored. The voluntary implications represent the actions the agent may or may not do in a given situation.

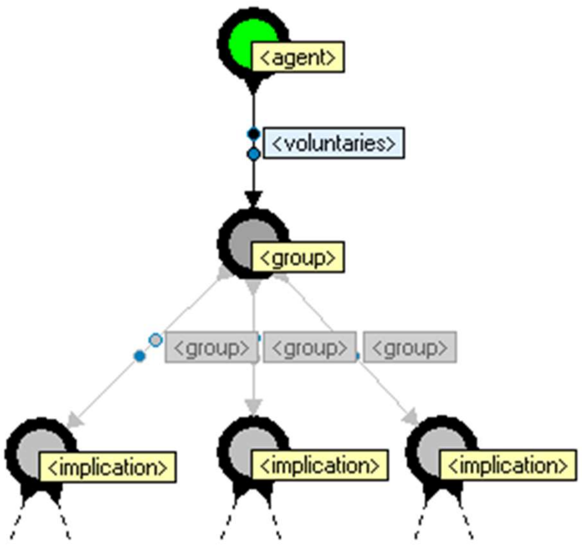


Figure 6.2 *An <agent> structure showing <voluntaries>.*

6.3 <involuntaries>

The <involuntaries>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts, and are classified as branch-implications. An involuntary implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the involuntary implication is ignored. The involuntary implications represent the agent's reflex actions that must be performed in a given situation.

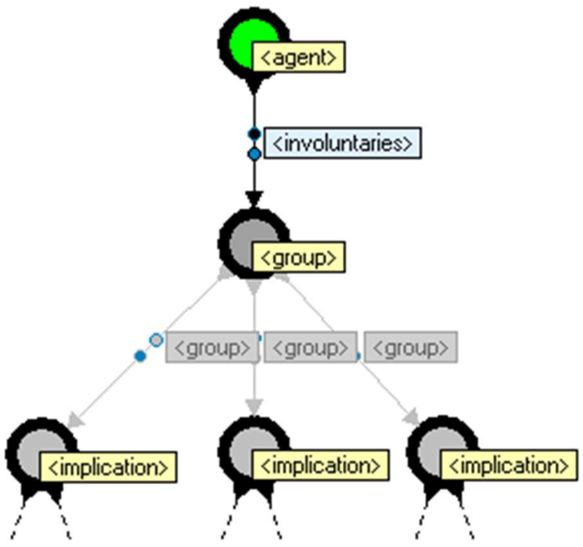


Figure 6.3 *An <agent> structure showing <involuntaries>.*

6.4 <constraints>

The <constraints>-group contains a set of <condition> structures. The AM uses these to determine whether a virtual context is permitted, and are classified as evaluation-conditions. The <condition> may either be a single condition or a double condition. If the constraint condition results in either <true> or <before>, then the virtual context is not permitted, and is removed from the VCT.

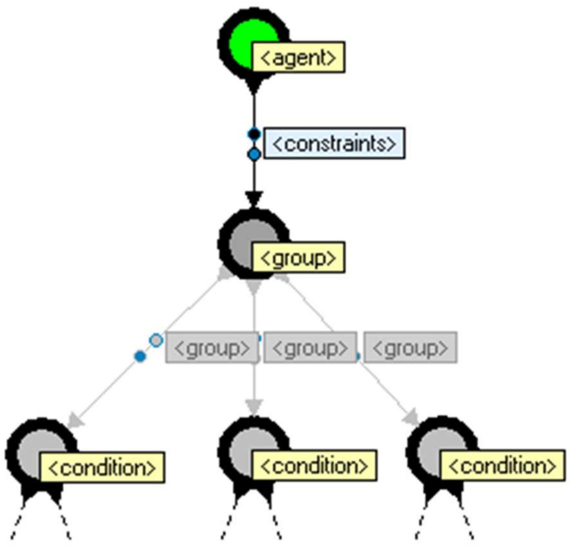


Figure 6.4 *An <agent> structure showing <constraints>.*

6.5 <consistencies>

The <consistencies>-group contains a set of <condition> structures. The AM uses these to determine whether a virtual context is consistent, and are therefore classified as evaluation-conditions. The <condition> may either be a single condition or a double condition. For a virtual context to be consistent, all consistencies conditions must either be <true> or <before>. If not, then the virtual context is removed from the VCT.

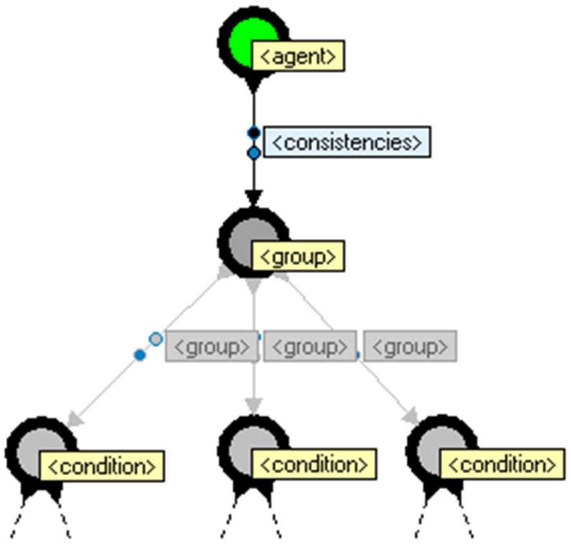


Figure 6.5 *An <agent> structure showing <consistencies>.*

6.6 <predictions>

The <predictions>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts, and are classified as branch-implications. A prediction implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the prediction implication is ignored. The prediction implications represent the agent's knowledge of how the "external world" will change in a given situation. Although prediction implications are branch-implications, they are converted to <wait> structures when used a part of an agent's inference.

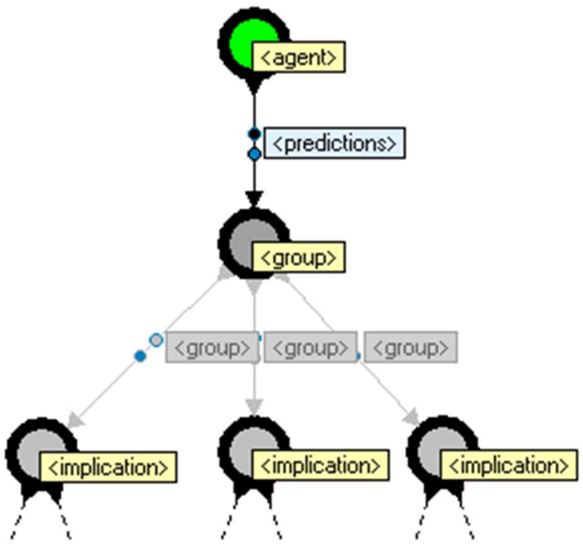


Figure 6.6 *An <agent> structure showing <predictions>.*

6.7 <preferences>

The <preferences>-group contains a set of <condition> structures. The AM uses these to determine how desirable a virtual context is, and are classified as evaluation-conditions. There are three different types of preference conditions, namely "<boolean>", "<count>" and "<obligation>". The type is stored in a vertex's label that is connected from the <condition> vertex with an edge labeled "<type>". When the <type> property is not present, then the type defaults to <boolean>. For boolean-preferences and obligation-preferences, the <condition> may either be a single condition or a double condition. For count-preferences, the <condition> must be a single condition.

The AM uses the <preferences> (together with <dislikes>) to rank the virtual contexts according to preference. For boolean-preferences, a virtual context has a high ranking if many conditions are either <true> or <after>. For count-preferences, a virtual context has a high ranking if many occurences of <true> situations (redexes) can be found. If obligation-preferences are present, then a virtual context is omitted from the ranking altogether if the condition result is either <before>, <both> or <false>. An agent's must achieve the obligations in any resulting inference.

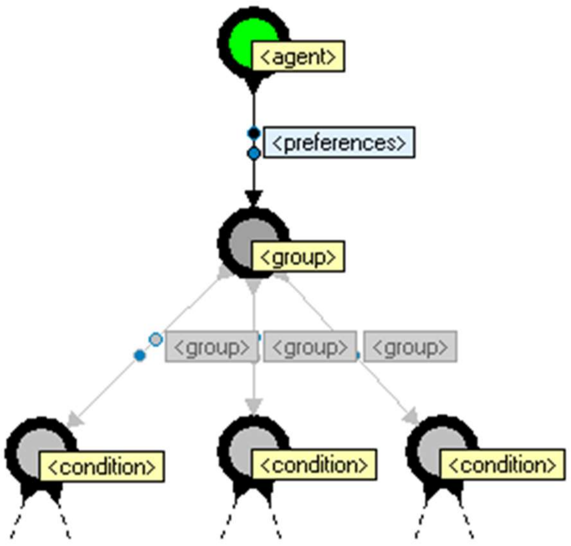


Figure 6.7 *An <agent> structure showing <preferences>.*

6.8 <dislikes>

The <dislikes>-group contains a set of <condition> structures. The AM uses these to determine how undesirable a virtual context is, and are classified as evaluation-conditions. The <dislikes> operate in the same way as <preferences>, except that <dislikes> make a virtual context have a low ranking. There are two different types of dislike conditions, namely "<boolean>" and "<count>". The type is stored in a vertex's label that is connected from the <condition> vertex with an edge labeled "<type>". When the <type> property is not present, then the type defaults to <boolean>. For boolean-dislikes, the <condition> may either be a single condition or a double condition. For count-dislikes, the <condition> must be a single condition.

The AM uses the <dislikes> (together with <preferences>) to rank the virtual contexts according to preference. For boolean-dislike, a virtual context has a low ranking if many conditions are either <true> or <before>. For count-preferences, a virtual context has a low ranking if many occurences of <true> situations (redexes) can be found.

6.9 The single-goal-agent

As an alternative to working with preferences and dislikes, a single goal is also supported. The single goal is represented as a <premise> and <conclusion> structure, that is connected directly to the <agent> vertex. The structure is therefore similar to a <condition> or <implication> structure, but is called the "goal-condition". When a single-goal-agent is run, it will first determine the value of the goal-condition. If the result is <before>, then the VCG is requested to build a VCT. The process will stop once an <after>-situation is found amongst one of the virtual contexts. The corresponding inference is then stored. The <before> situation is therefore the condition of the agent, and the <after> situation is the goal of the agent. The single-goal-agent can be used as a simple problem solver.

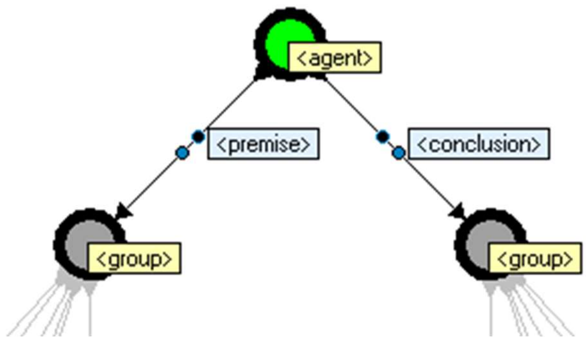


Figure 6.8 *The single-goal-agent.*

6.10 The agent types

There are two types of agents, namely "<active>", and "<solve>". The type is stored in a vertex's label that is connected from the <agent> vertex with an edge labeled "<type>". An active-agent will calculate and execute an inference. A solve-agent calculates an inference, but does not execute it. The result of an active-agent is "<achieved>" if the inference could be calculated and if the inference executed successfully. In all other cases the result is "<failed>". The result of a solve-agent is "<achieved>" if the inference could be calculated. In all other cases the result is "<failed>". A single-goal-agent has additional results "<both>" and "<neither>". These values are returned in the cases where the goal-condition results in "<both>" or "<neither>", respectively, because in these cases no attempt is made in calculating the inference. If the goal-condition is <after>, then no inference is calculated and the agent's result is simply <achieved>.

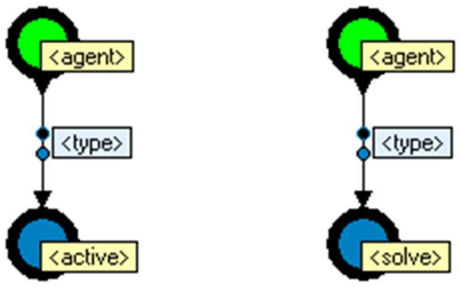


Figure 6.9 *An active-agent and a solve-agent.*

6.11 An example of an agent

To illustrate the use of an <agent>, an example is shown in the form of a simple problem solver. The puzzle is a well-known puzzle that is described as follows:

A farmer finds himself on one side (left side) of a river with a chicken, a fox, some grain and a boat. The farmer must take all these items to the other side (right side) of the river using the boat. The rules state, however, that the farmer may only take zero or one item in the boat. Furthermore, the fox and the chicken must not be left unattended, and neither should the chicken and the grain. What is the minimum order of actions that the farmer must perform to move himself and all the items to the right hand side?

We can quite easily solve this puzzle ourselves. There are two possible solutions:

<u>Solution 1</u>

1. Farmer takes chicken to right side.
2. Farmer travels to left side.
3. Farmer takes fox to right side.

4. Farmer takes chicken to left side.
5. Farmer takes grain to right side.
6. Farmer travels to left side.
7. Farmer takes chicken to right side.

<u>Solution 2</u>
1. Farmer takes chicken to right side.
2. Farmer travels to left side.
3. Farmer takes grain to right side.
4. Farmer takes chicken to left side.
5. Farmer takes fox to right side.
6. Farmer travels to left side.
7. Farmer takes chicken to right side.

To solve the problem in OutOfBrain, we start by making a model of the situation in the form of a graph. Here we describe the concepts of the puzzle that are relevant to the problem proposed. This will be our application graph (as opposed to the instruction graph).

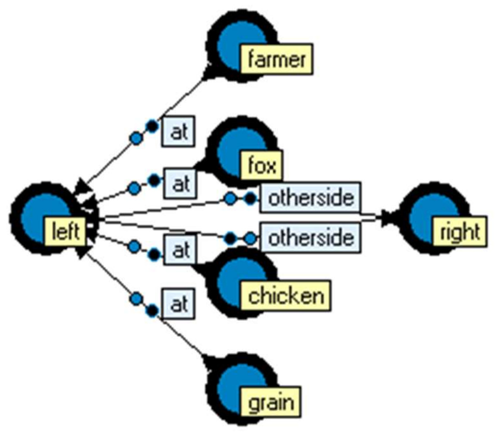


Figure 6.10 *The appligation graph.*

The application graph shows a simplified view of the first two sentences of the puzzle. Notice that the graph could have been named and structured differently, and still provide a good graph to work with. A "good graph" is loosely defined by two criteria:

1. "A graph that describes the essence of the static situation in its simplest form".
2. "A graph where relevant changes in the situation can be described in terms of simple graph operations".

A more formal definition of a "good graph" is difficult to determine because the two criteria show a trade-off relationship. Notice that we chose not to represent the "boat" in the graph, as it is not relevant to the essence of the puzzle. The boat is always with the farmer, so we do not need to represent the boat. Similarly, we do not have to represent the river. The puzzle is simply about a farmer, three distinct items and two distinct sides. It is easy to see why we need the "at" edges, because we need to be able to describe where the farmer and the items are. But one could argue that we could have left out the two "otherside" edges, because we can still describe the situation without them. The two "otherside" edges are there to meet the demands of the second criterium, for reasons that will become apparent later on. Constructing a good graph is usually an iterative process, tweaking the application graph during the design of the instruction graph. Making a good graph straight from the start requires practice with the type of graph rewriting used in OutOfBrain. The criteria for a good graph can be further extended with more requirements, such as "flexibility" and "re-usability", but these topics are beyond the scope of this text.

The next step is to create the instruction graph. The following paragraphs will describe each component of the instruction graph. Figure 6.11 shows a zoomed out view of the whole graph.

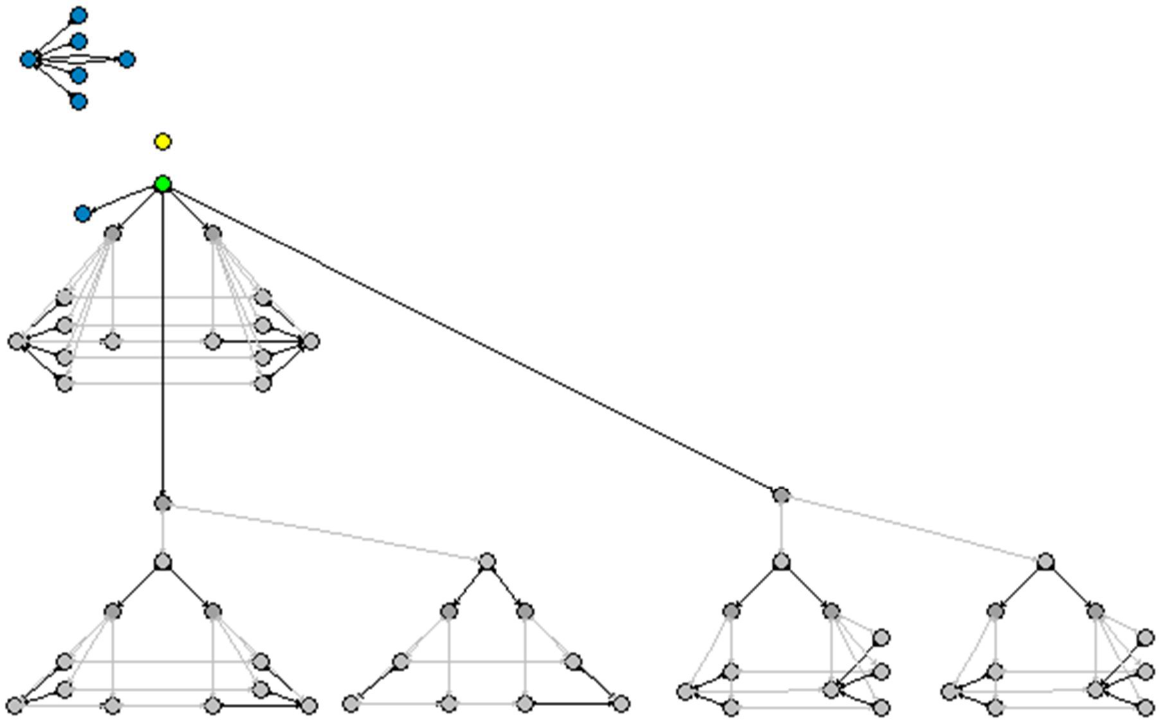


Figure 6.11 *Overview of the puzzle and the agent.*

In the zoomed out graph the application graph can be seen in the top left corner. The main <agent> structure is shown directly below the application graph, shown in figure 6.12. The agent is a solve-agent, as shown with the <type> edge. The agent is a single-goal-agent as it has <premise> and <conclusion> groups. The goal-condition shows that the premise matches the application graph, and therefore the goal-condition is in the <before> state. The agent would therefore attempt to calculate an inference if it were executed. Notice how the transition from the premise situation to the conclusion situation corresponds with what the puzzle asks us to solve.

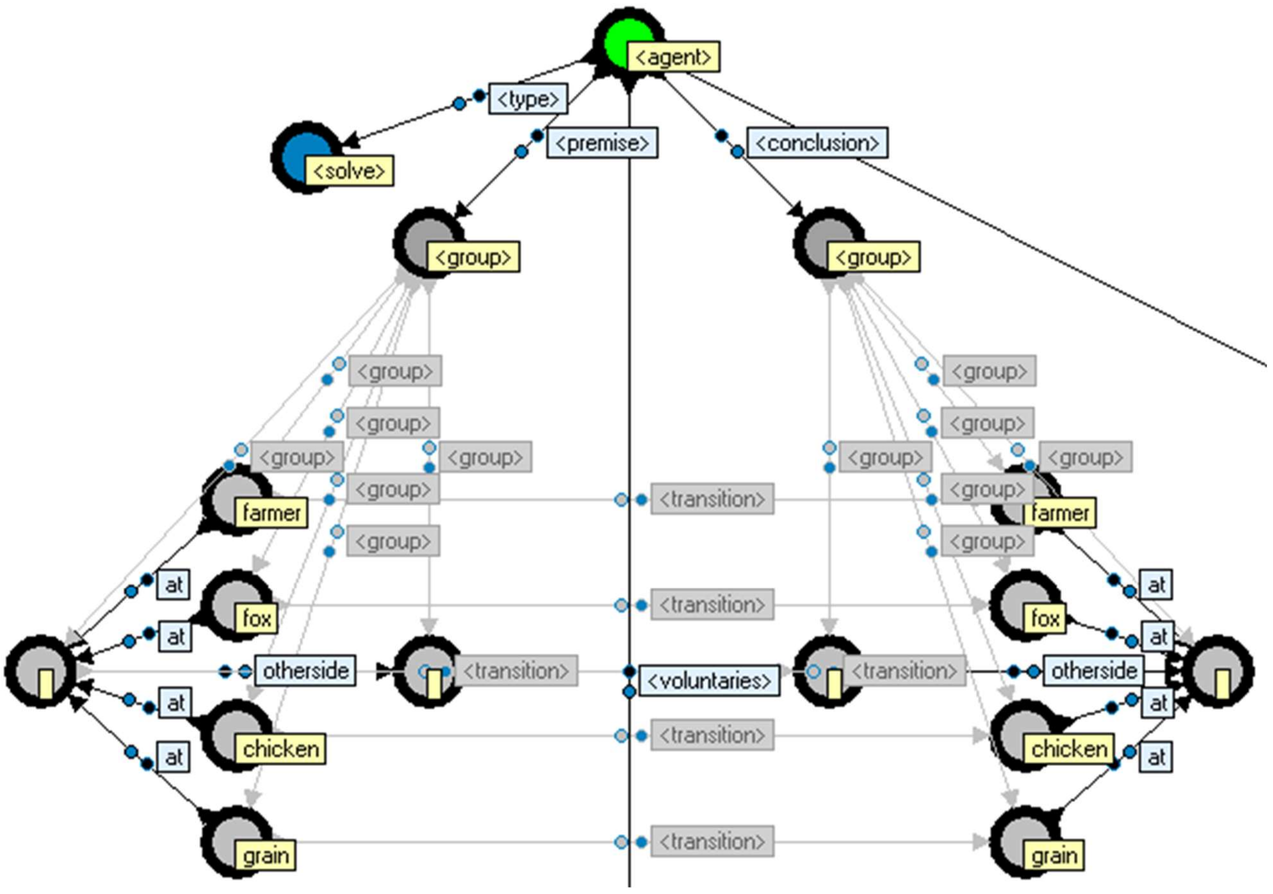


Figure 6.12 *The agent.*

The puzzle allows the farmer to go to the other side with just one item, or with no item at all. These two possible actions can be described as two voluntary implications. Figures 6.13 and 6.14 shows these voluntary actions respectively.

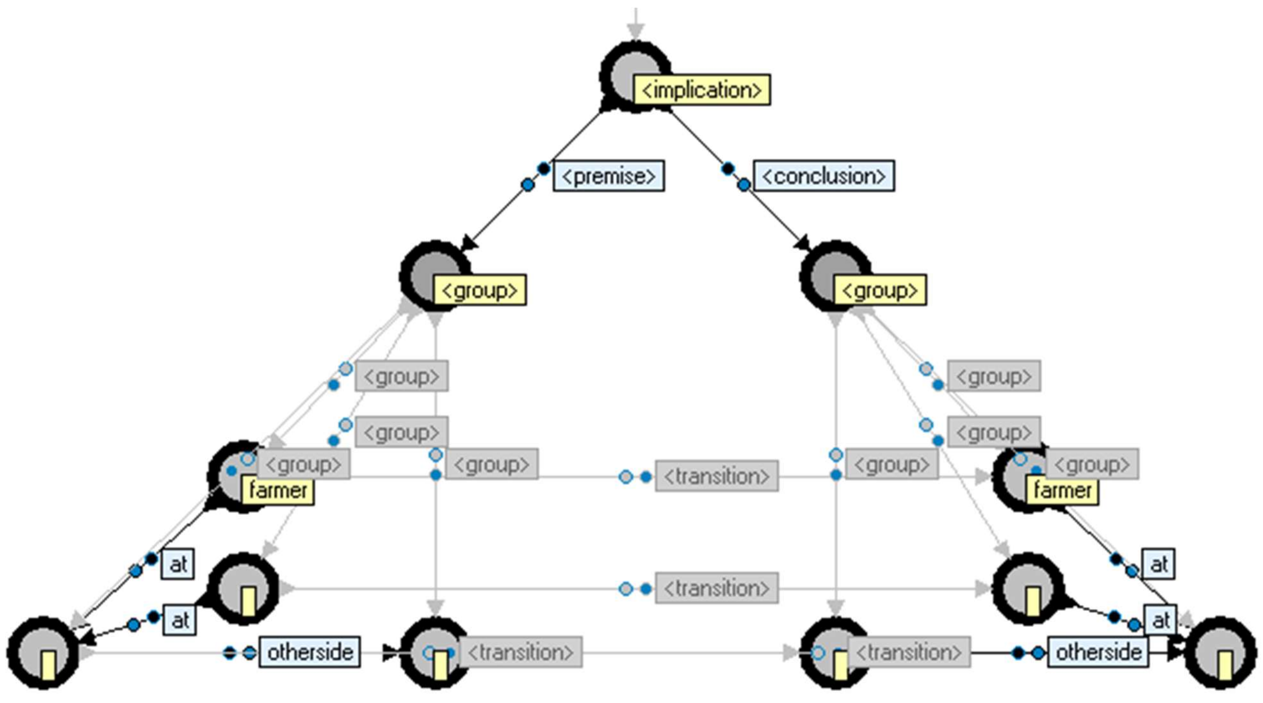


Figure 6.13 *Voluntary implication #1.*

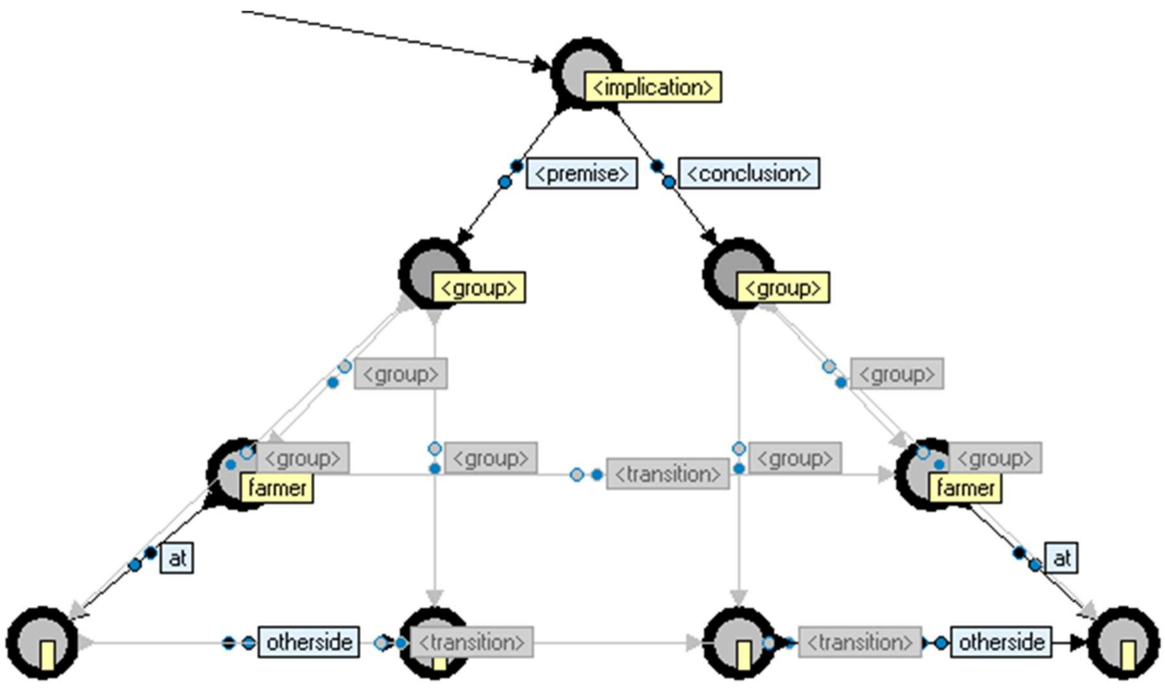


Figure 6.14 *Voluntary implication #2.*

We have almost described the whole puzzle to the agent. The two remaining rules are two constraints, which state that the chicken and fox should not be left unattended, and neither should the chicken and the grain. We can describe these two constraints as constraint conditions. We need to describe these constraints as conditions in such a way that the <true> or <before> situation describes the constraint. Figures 6.15 and 6.16 shows how the before situation describes the absence of the farmer.

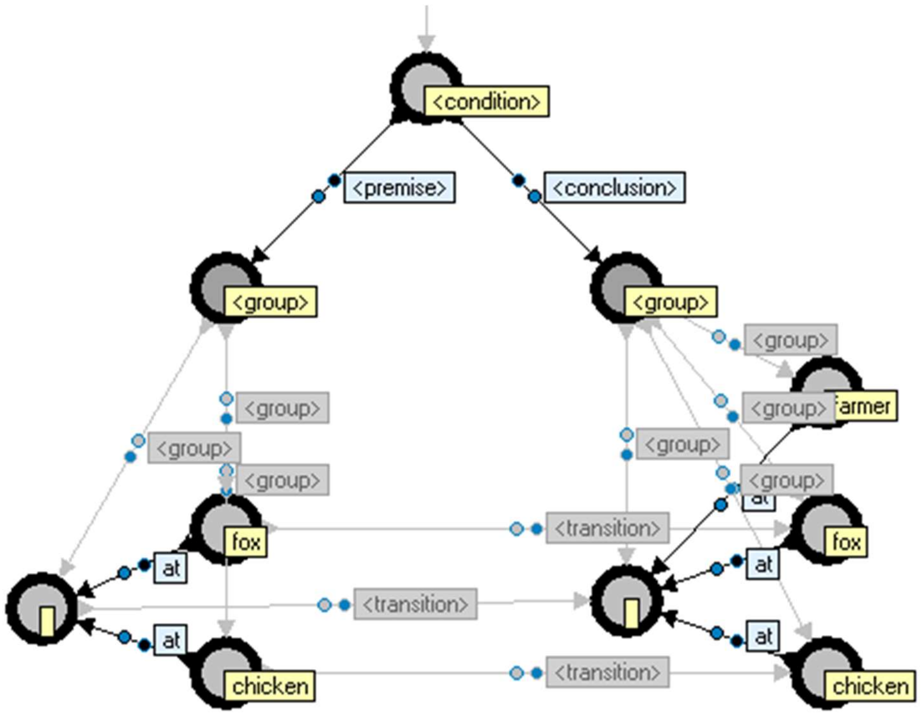


Figure 6.15 *Constraint condition #1.*

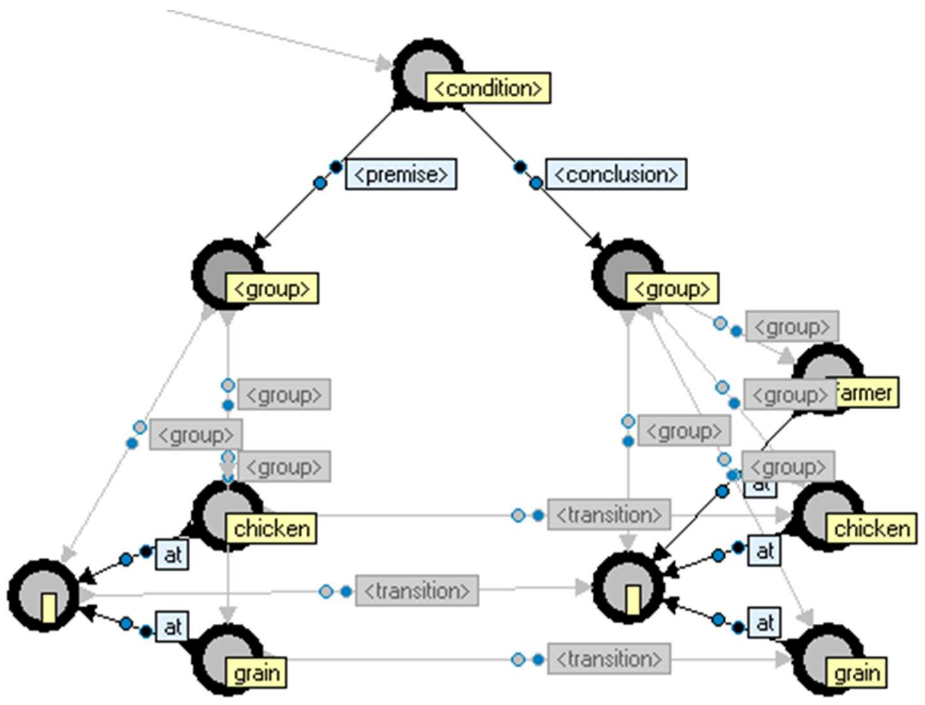


Figure 6.16 *Constraint condition #2.*

Representation of the entire puzzle is now complete. The agent has enough information to calculate an inference. If the agent is executed, then the agent will try many combinations of the voluntary implications to see which sequence will produce the desired result. When the inference has been found, the agent will print the sequence of instructions (implications) that were necessary to achieve the goal. And inference edge is connected from the <agent> vertex to the first instruction of the inference. As this is a solve-agent, the inference is not executed. The result is shown in figure 6.17. The instructions are shown from left to right, next to the <agent> vertex. Although not all visible, there are seven implications that describe the actions made by the farmer, which directly corresponds with one of the seven-step answers provided in solution 1 and solution 2. In fact, the agent is capable of finding both solutions, but just one of them chosen at random for the inference.

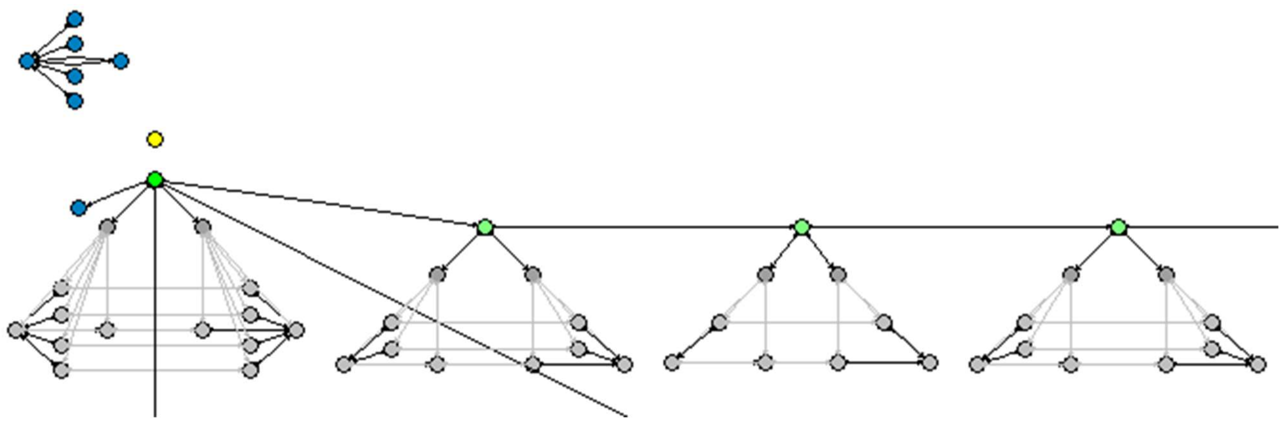


Figure 6.17 *Overview of the graph after execution of the agent.*

Figure 6.18 shows a closer view of the first instruction. It is visible that this resembles the first voluntary implication, except that the labels for "chicken", "left" and "right" are filled in. The agent does this ensure that the correct item is chosen.

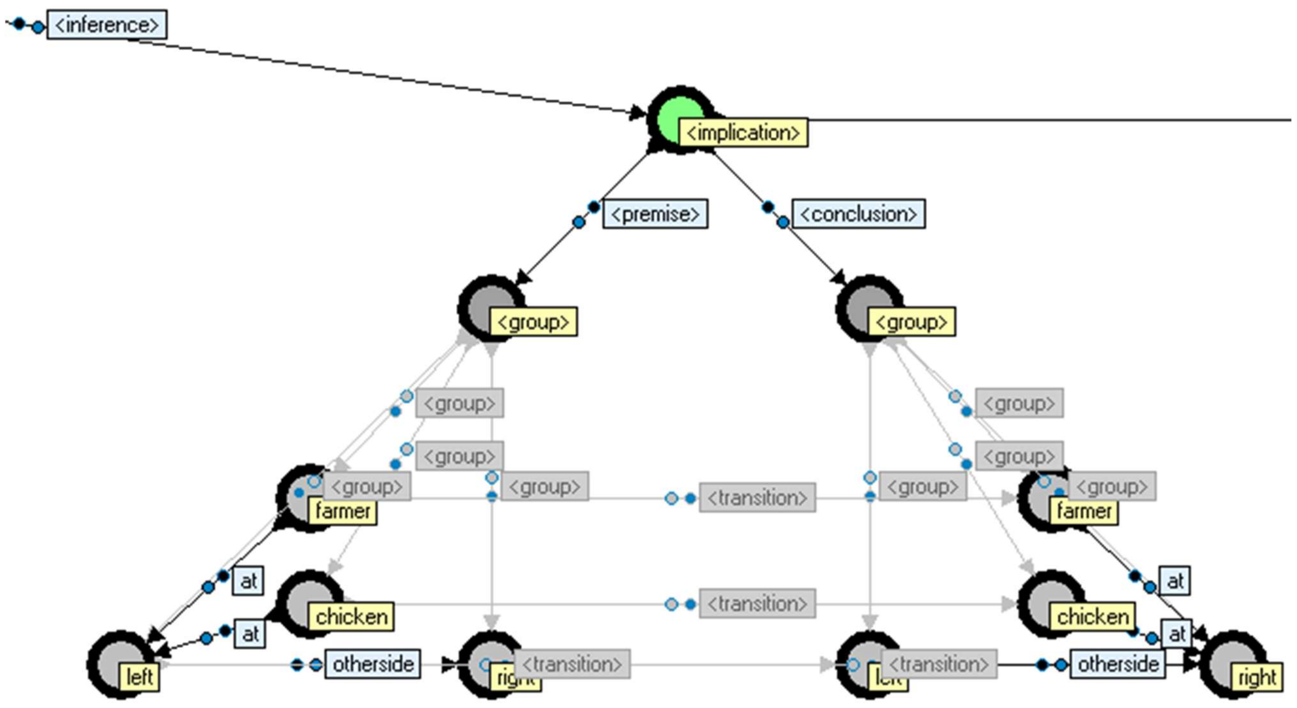


Figure 6.18 *The first instructions of the inference.*

If the agent were an active agent, then the inference would have been executed, moving the farmer with the items to the other side according to the instructions in the inference.

7 TCP communication

OutOfBrain supports structures for communicating via TCP. There are two basic structures for TCP, namely "<tcpserver>" and "<tcpclient>". They are not classified as "instructions" because they do not require a <process> to execute them. Instead, the TCP structures are run directly by the TCP manager. The following two chapters describe these instuctions.

7.1 <tcpserver>

The <tcpserver> structure is interpreted as a listening TCP server. Many <tcpserver> structures can be created, providing that each server listens to a different "port". A <tcpserver> structure is shown in figure 7.1. When the <status> property is set to "<enabled>" then a TCP server is created that listens to the port number that is set in the <port> property. If the creation of the server was not successful, then the <status> property is automatically set to "<disabled>". The server can be disabled by setting the <status> property to "<disabled>". The <status> property may be set by implications in the instruction graph.

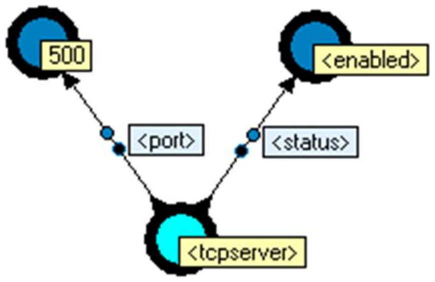


Figure 7.1 *The <tcpserver> structure.*

When a client connects to a <tcpserver>, a <client> structure is connected as shown in figure 7.2. The <address> property of the <client> shows the client's address. The <status> property of the <client> is set to "<connected>" while the connection persists. The <client> can be disconnected by setting the <status> property to "<disconnected>". When a <client> disconnects, the <client> structure is removed. Many clients may connect, and for each connection is <client> structure is created.

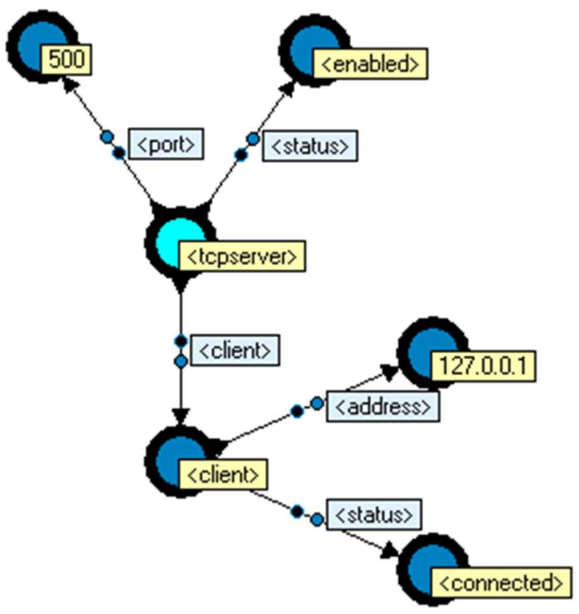


Figure 7.2 *A <tcpserver> structure with a connected client.*

7.2 <tcpclient>

The <tcpclient> structure is interpreted as a TCP client. Multiple <tcpclient> structures can be created. A <tcpserver> structure is shown in figure 7.2. When the <status> property is set to "<connect>" then a TCP client is created that connects to an address that is set in the <host> property, with the port number that is set in the <port> property. If the connection could be establisched successfully, the <status> property is set to "<connected>", else it is set to "<disconnected>". The client can be disconnected by setting the <status> property to "<disconnected>". The <status> property may be set by implications in the instruction graph.

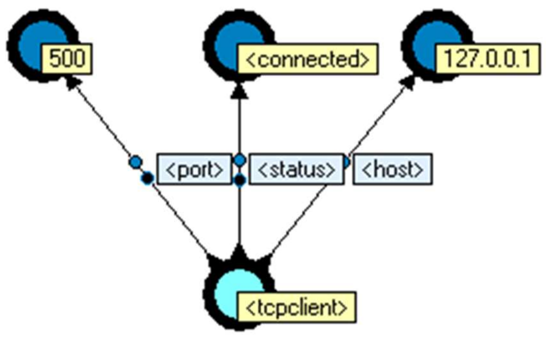


Figure 7.3 *The <tcpclient> structure.*

7.3 Sending and receiving data

The <tcpserver> and <tcpclient> both send and receive data in the same way. To send data from a <tcpclient> to its connected host, first create a string of vertices that are interconnected by edges labeled "<next>", directing from the start to the end of the string to be sent. In the labels of each vertex of this string, enter a piece of text. The last vertex in the string should be labeled "<newline>". Next, connect an edge from the <tcpclient> to the first vertex of the string and label it "<send>". At this moment, the TCP manager will detect the presence of the <send> edge, and will determine whether a string of veryices is connected that ends with a <newline> vertex. Then the string will be removed, and the textual version is sent to the host. The data that is actually sent is a string of texts, separated by space characters, and ending wit a #10 byte character. Similarly, if a <send> edge is connected from a <tcpserver>'s <client> vertex to such a string of vertices, then this text is sent to the corresponding client.

When a <tcpclient> receives data from its connected host, it collects all data until a #10 character is read. The data is interpreted as text, and is separated into smaller pieces of text where whitespace is used as a delimiter. These pieces of text are then stored into the labels of a string of vertices. An edge labeled "<received>" is then connected from the <tcpclient> vertex to the first vertex of the string. Similarly, a <client> of a <tcpserver> receives data in the same way.

The easiest way to illustrate this, is to create a <tcpserver> structure and a <tcpclient> structure in the same graph, and let the <tcpclient> connect to the <tcpserver>, as shown in figure 7.4. The figure shows that the client has received a string of data. The data can be sent back to the server by renaming the "<received>" edge to "<send>".

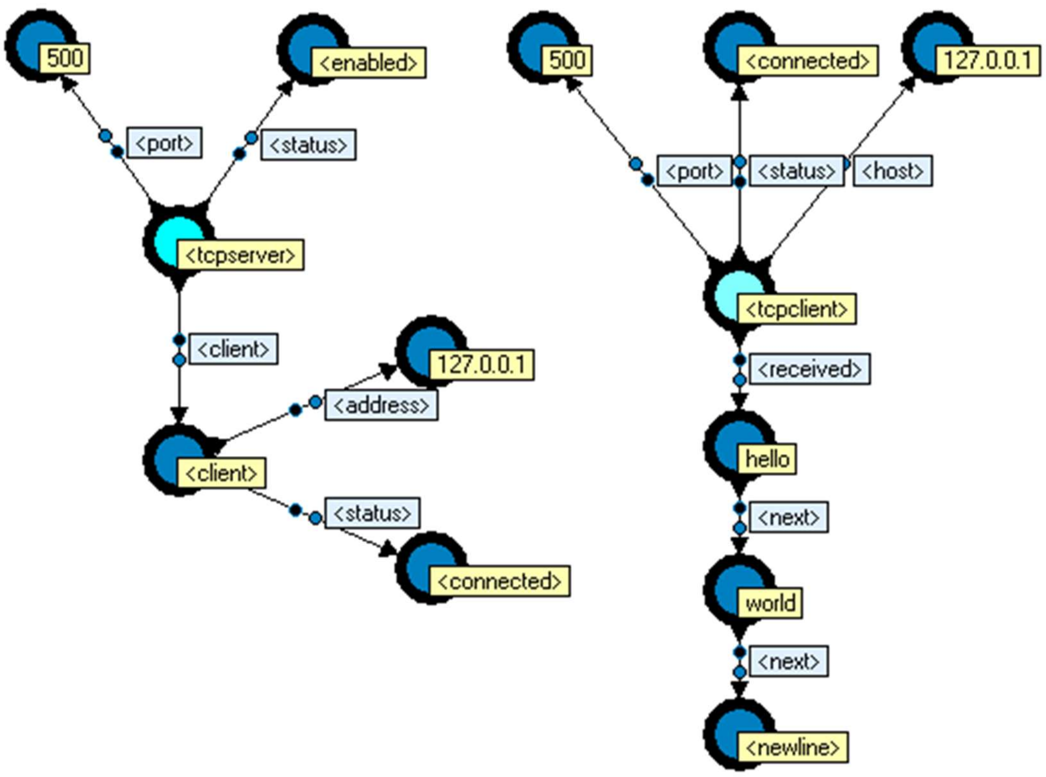


Figure 7.4 *A <tcpclient> connected via TCP to a <tcpserver>.*