4
The Graph Machine

4.1.Architecture

The Saint Thomas Aquinas Machine algebraic specification described in Appendix A is mapped in object oriented architecture. Comparing to the algebraic specification, commands and registers were added. Some commands were removed. Design patterns [9] ware used intensively. This architecture is implemented in Java and distributed as a standalone application. It also provides an API to help integration with other applications. The project is intended to be a generical solution; with modularity, low coupling, and robustness.

The machine's primitive type is graph. Its memory stores only graphs, this means that the programs are represented by graphs as well. Because of that, the machine's theoretical model describes a non-deterministic machine. The architecture presented needs to use some criteria for the choice involved. This choice is modularized so it can be changed easily. This is shown below.

The architecture is divided over two components: the machine's memory and the machine's processing unit. The memory is called The Order of Preachers Inference Model. It is a graph manipulation component that holds the machine's business model (in an MVC [9] cfr. section 1.2 sense). The processing unit is called Saint Thomas Aquinas Machine. This component holds the fetch-decode-execute cycle of the machine and also holds the implementation of the commands of the machine's assembler language.

The machine architecture was influenced by MVC (Model-View-Controller Architectural Pattern). The pattern divides the architecture in tree layers: model, controller and view. Model and view layers do not interface with each other; all of its interaction is done through the third layer. The model layer in MVC is the layer responsible for data access and knowledge of business rules.

Figure 1 shows the two components [18] of the machine, its interfaces and interactions with external classes and artifacts. The shown interfaces conform to design patterns [9]. Four design patterns where used in the interfaces: factory method, singleton, façade and observer.

The interfaces in Figure 1 are:

- *machine factory* and *inference model factory*: part of the patterns factory method and singleton
- *Manager* and *facade* are façades
- *Choice listener* is part of the design pattern observer.

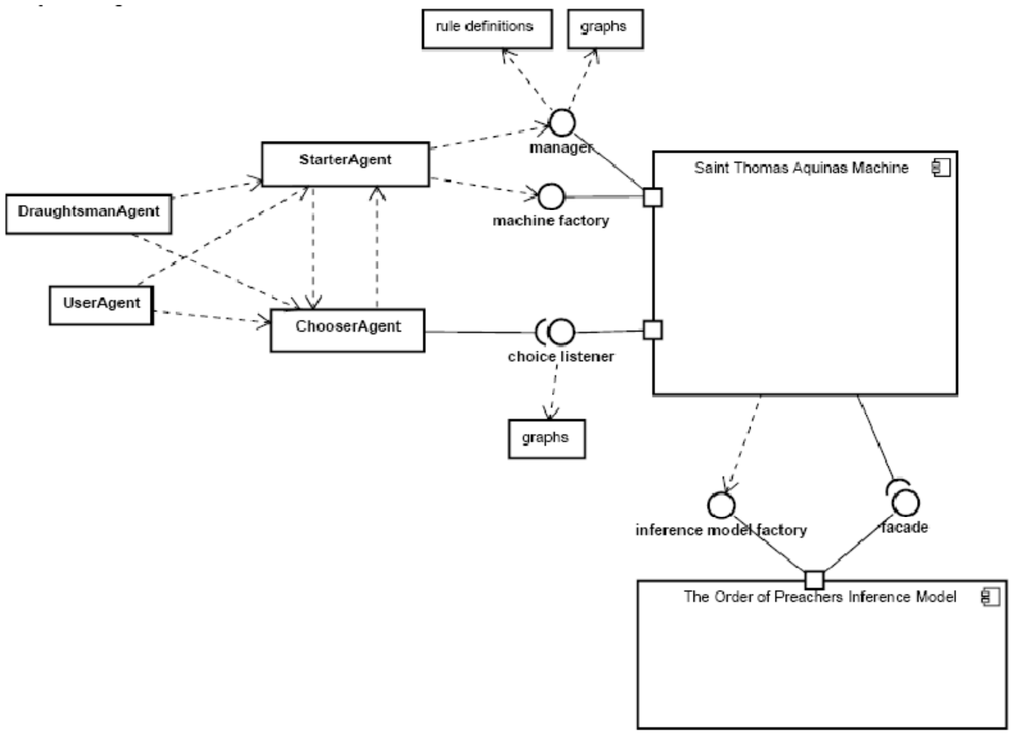

Figure 1 Saint Thomas Aquinas Machine Component Diagram

The machine and the inference model interact trough the interface *facade* provided by the inference model. This component provides only one port for interaction with other components. The port holds the *facade* interface and the inference model *factory* interface. When a machine is build, it invokes the inference model *factory* to create an instance of the model that will serve as its memory.

The Saint Thomas Aquinas Machine is the component that exposes interfaces for interaction with the world outside the machine. It contains two ports. The first port holds *manager* and *machine factory*. It is similar to the inference model's port. Its interfaces are used for component instantiation and interaction. The second existing port holds *choice listener*, an implementation of the design pattern observer, which treats special agent-machine interaction.

Figure 1 also contains external classes and artifacts that explain the possible uses that can be made of the machine. The *StarterAgent* class is responsible for the basic interaction. It invokes the *factory* to instantiate a new machine; it inserts into the machine the artifacts *rule definitions* and *graphs* and it orders the machine to be started.

The *ChooserAgent* class does the special interaction with the machine. The machine's execution can be viewed as a game from game theory [19]. Every round a rule or a set of rules is applied to the formula to be proved (the goal). The rules can be well succeeded and generate new formulae or not. After the round it is time for a decision. Other rules should be applied to the same goal or the new goals. The chooser agent makes these decisions along the game. It contains the strategy desired to be used to prove the formula.

The Component Diagram also shows two other classes: *DraughtsmanAgent* and *UserAgent*. These agents have an auxiliary role in the agency. The first integrates the machine with drawing systems to draw the proof. It can make a drawing of it in the end of the proof or at every round. The interfaces manager and choice listener return, up on invocation, graphs artifacts with the proof to be drawn. The second class integrates the machine with a human-computer interface that can either be a command prompt or a GUI.

4.2.Functionality

This section describes the functionality of the graph machine. This virtual machine interprets an assembly language designed for it – described in section 4.3. The architecture bellow contains many details. It was done in this way because of the project decision towards a generical solution; with modularity, low coupling, and robustness. The architecture also is influenced by the existing Algebraic Specification (Appendix A), that led the solution to use the presented registers.

In order to make a theorem proving platform based on graph transformations; more than graph transformations is needed. The complexity involved in theorem proving with strategies can better be accomplished by a directed approach that led the specifier to the presented machine with its registers and its running cycle.

In order to interpret the assembly commands, the machine runs through a fetch-decode-execute cycle just as any physical processor does with byte code. With this architecture, it is possible to migrate the machine from this assembly-oriented implementation into a byte-code-oriented implementation if the change is a tuning. It also gives the program good levels of maintainability, modularity and elegance.

As any physical machine, Saint Thomas Aquinas Machine has some registers and is connected to a memory. Its internal architecture is illustrated as a block diagram (just as any CPU would be) in Figure 2. The machine's memory is The Order of Preachers Inference Model described as a component in section 4.1 and shown as a large block with a façade block on its interface and other blocks on the inside.

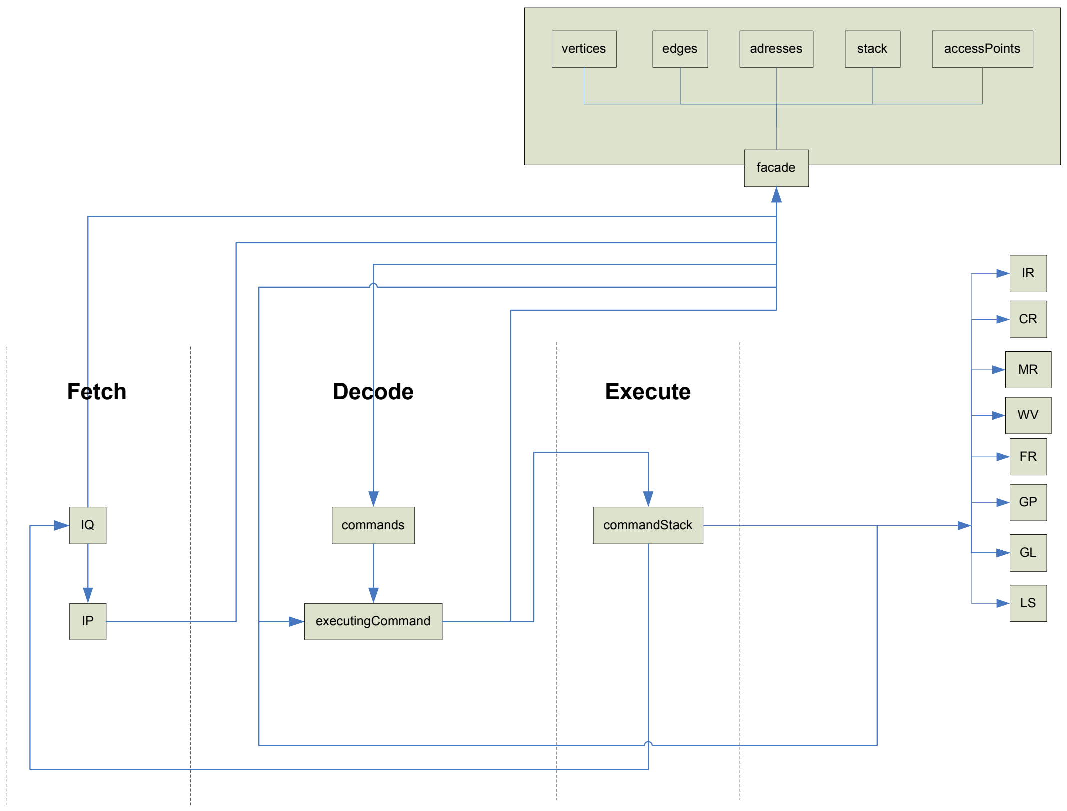

Figure 2 Saint Thomas Aquinas Machine Block Diagram

The machine's *memory* works with graph vertices, with graph ordered edges between two given vertices and with strings just as formalized in Appendix A. As it is shown in Figure 2, the *memory* holds some sets: *vertices*, *edges*, *addresses*, a *stack* and *accessPoints*. The *vertices* and the *edges* are sets of graph primitive types. *Addresses* is a set of pointers to these objects or to strings used as variables. *Stack* is a stack data-structure that can hold vertices, edges and strings. *AccessPoints* is a set of vertices.

The machine's *memory* stores a large graph of data and instructions. The data can be indexes in the set *vertices*, in the set *edges* or in the set *addresses*. The machine's instructions are stored together with the data. The programs starting points are indexed in a set called access points. Following de edges, the machine is able to execute the instructions contained in the vertices. The memory also has a stack in which its registers can be stored for a change in the execution context.

The machine registers are:

- The *Instruction Queue* (IQ): A queue data structure where the references for the next commands are enqueued.

- *Instruction Pointer* (IP): A reference to the command currently being processed.
- *commands*: This register holds the definitions of the machines commands (described in section 4.3) and the string that represents them in the assembly language.
- *executingCommand*: it's the register that holds the currently executing command and its code.
- *commandStack*: it executes the commands and as instrumentation, it is able to undo commands that were executed.
- *Interupt Register* (IR): a boolean value that interrupts the machine cycle and waits for a decision. It makes the machine semi-automatic and strategy-based.
- *Compare Register* (CR): a boolean value used by the comparison and jump commands.
- *Match Register* (MR): holds the string found by a match command.
- *Working Vertex* (WV): an auxiliary register used to hold a vertex reference.
- *Function Register* (FR): holds the return of a function.
- *Goal Pointer* (GP): The program current goal. It supports strategy-oriented programs.
- *Goal List* (GL): The list of goals on hold. It is a list in order to be ranked by the DeciderAgent (described in section 4.1). It supports strategy-oriented programs.
- *Local Strategy* (LS): a queue of rules to be executed. It supports strategy-oriented programs.

The fetch-decode-execute cycle depends on these registers. In the *fetch* process, the machine gets from the IQ or from the memory's *accessPoints*, the next instruction's reference and moves it to the IP. The decode phase consist of getting the contents of the vertex referenced by the IP, finding its code in *commands*, moving it to the *executingCommand* and adding the machine's current state to it (by coping the registers as its internal state). *Execute* is where the command is executed by the *commandStack* and where the changed registers are updated.

The commands store the machine’s state in its attributes. In this way, after execution, SaintThomasAquinasMachine can refresh its state according to the new state stored in the command class. It is not the design pattern state; the machine’s state corresponds to the different contents of its registers. Whenever a decision is needed, the machine interrupt its execution in after execution, right before fetching a new command.

4.3.Assembler Language

The Saint Thomas Aquinas Machine receives as input an archive with a program in an assembly-like programming language. This section is intended to describe this programming language. Syntax errors can be found by applying the machine's parser to the input program.

First of all, it is important to notice that the machine is graph oriented. Its memory is divided into vertexes and edges. The program is a graph as well. Its instructions are vertices' contents. The machine simulates non-determinism. It works based on an instruction queue. Thus, an instruction (vertex) can have more than one next instruction (vertices connected by a directed edge).

Vertices syntax – observe the presences or absences of spaces - is as follows:

```
id#content
```

- id: A unique identifier to this object (for any vertex, edge or pointer), the vertex will be repeated in the program, the id will distinguish it.
- content: The vertex's content. For program instructions we may use:

  instr arg1,arg2,arg3,arg4

  - instr: Instruction (described below).
  - Arg1: First Argument (depends on the instruction).
  - Arg2: Second Argument (depends on the instruction).
  - Arg3: Third Argument (depends on the instruction).
  - Arg4: Fourth Argument (depends on the instruction).

The edge syntax is as follows:

```
vertex1$id#order,content$vertex2
```

- vertex1: Source vertex.
- vertex2: Target vertex.
- id: A unique identifier to this object (for any vertex, edge or pointer), the edge should not be repeated in the program.

- order: an integer number for reference (obligatory, -1 means unordered and -2 cannot be used).
- content: The edge's content (optional).

A simple program looks like this:

```
C1#vcreate V1 $e1,-1$ C2#vcreate V2
C2#vcreate V2 $e2,-1$ C3#vcreate V3
C3#vcreate V3 $e3,-1$ C4#ecreate V1,E1,V2
C4#ecreate V1, E1, V2 $e4,-1$ C5#ecreate V1,E2,V3
C5#ecreate V1, E2, V3$e5,-1$ C6#sto V1,A
C6#sto V1, A $e6,-1$ C7#sto V2,B
C7#sto V2, B $e7,-1$ C8#sto V3,C
```

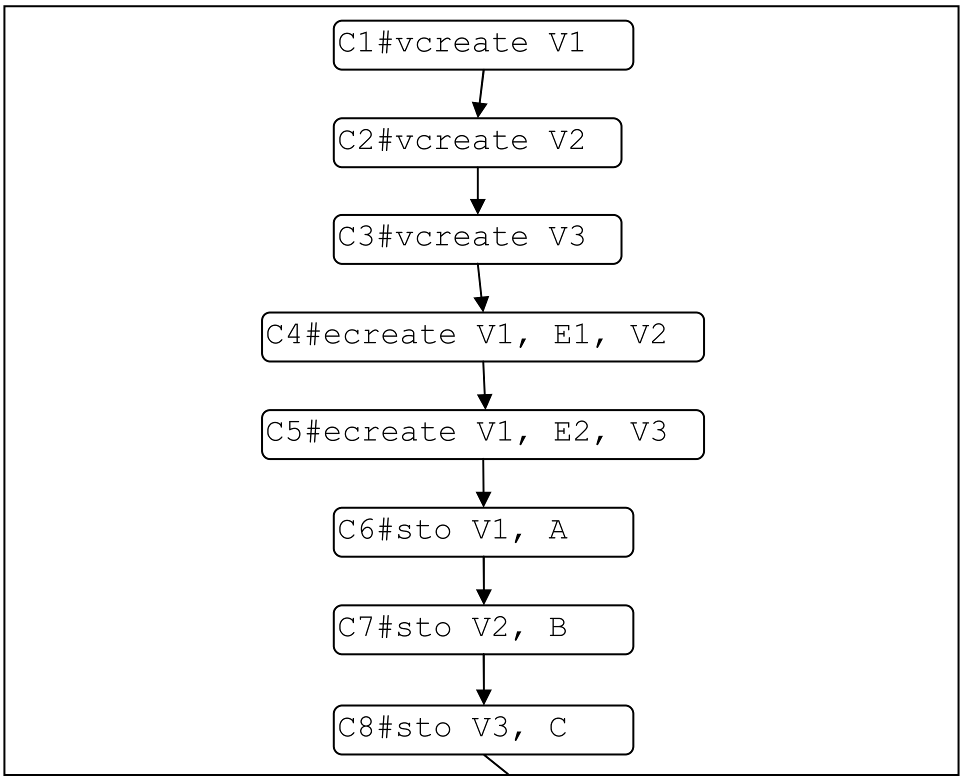

Figure 3 Simple program

Vertices can have more than one edge associated, as the example:

```
C8#sto V3, C $e8,-1$ C9#screate S
C9#screate S $e9,-1$ C10#screate SAB
C10#screate SAB $e10,-1$ C11#sedges S, V1
C11#sedges S, V1 $e11,-1$ C12#sreset S
```

```
C12#sreset S $e12,-1$ C13#jse S
C12#sreset S $e13,-1$ C13b#jsne S
C13b#jsne S $e14,-1$ L#snext S,E
L#snext S,E $e15,-1$ C14#target V,E
C14#target V,E $e16,-1$ C15#sadd SAB,V
C15#sadd SAB,V $e17,-1$ C13#jse S
C15#sadd SAB,V $e18,-1$ C13b#jsne S
C13#jse S $e19,-1$ Laddr#mcreate M,B
Laddr#mcreate M,B $e20,-1$ C17#match SAB,M
C17#match SAB,M $e21,-1$ C18#sto V,MR
```

…

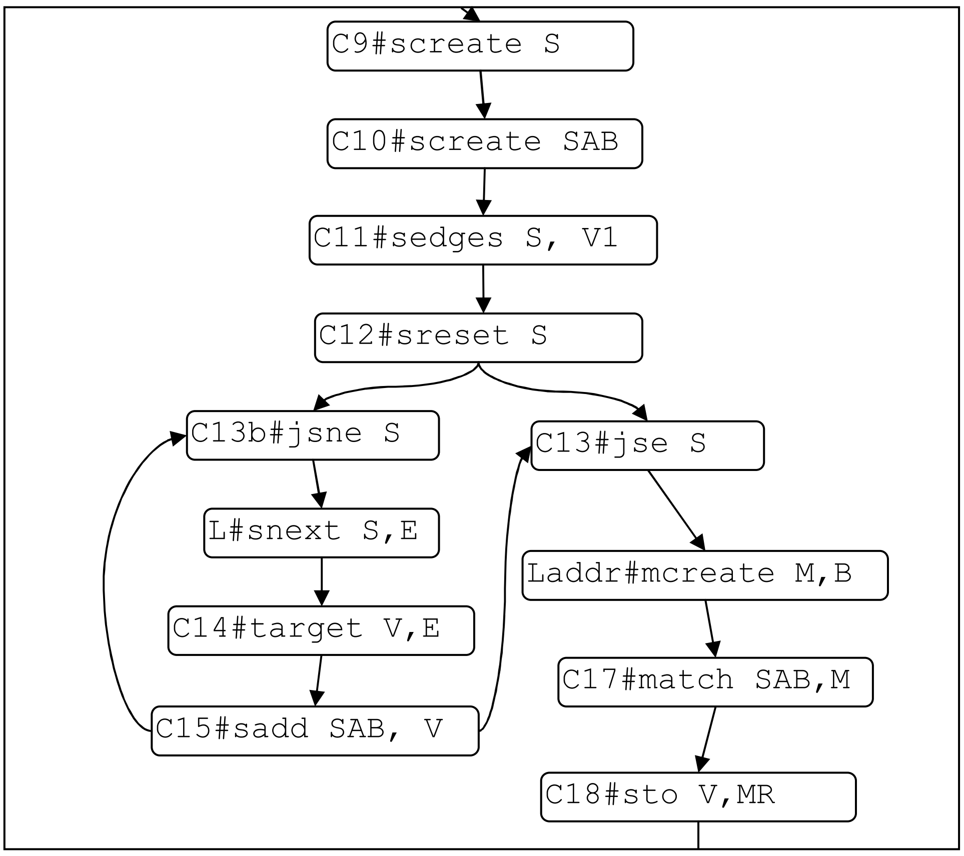

Figure 4 Code Example

Every instruction must be connected to the graph starting at that vertex to be executed. Bellow, the instructions are described with its parameters with its functionality and its assertives. If an assertive is not respected, the instruction is ignored.

Command: vcreate

Parameters:

Vvar - > Id of the vertex to be created.

Syntax: vcreate VVar

Creates a vertex whose id is VVar. This vertex can be referenced in the rest of the program through this id. If the id already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: uvcreate

Parameters:

Addr - > Address of the pointer to be created.

String - > content (opcional).

Syntax: uvcreate Addr [, String]

Creates a vertex whose id is unspecified and a pointer (Addr) to this vertex. If content is provided, it will be set as vertex's content. This vertex can be referenced in the rest of the program through this pointer. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: ecreate

Parameters:

VVar - > Source vertex.

EVar - > Edge id.

VVar - > Target vertex.

Syntax: ecreate VVar, EVar, VVar

Creates an edge whose id is Evar and whose extremities are the first and the second VVar parameters, respectively origin and destination. The source and target vertex are obligatory, if not provided or if the vertices do not exist, the edge will not be created. This edge can be referenced in the rest of the program through its EVar. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: uecreate

Parameters:

(VVar or WV) - > Source vertex (id or Working Vertex register).

(VVar or WV) - > Target vertex (id or Working Vertex register).

order - > Order integer.

String - > content (optional).

Syntax: uecreate (VVar or WV), (VVar or WV), order [, String]

Creates an edge whose id is unspecified and whose extremities are the first and the second (VVar or WV) parameters, respectively origin and destination. The source and target vertex are obligatory, if not provided or if the vertices do not exist, the edge will not be created. The parameters order and content will be set to the created edge.

Command: eorder

Parameters:

EVar - > Edge id.

order - > Order integer.

Syntax: eorder EVar, order

The parameter order will be set to the edge whose id is EVar.

Command: sto

Parameters:

Addr - > Address.

(String or MR) - > Content (String or Match Register).

Syntax: sto Addr, (String or MR)

Store places string as content of the vertex, edge, match or pointer whose id was passed in Addr. The Addr must be valid.

Command: pcreate

Parameters:

Addr - > Address of the pointer to be created.

(VVar, EVar, Addr, GP or WV) - > object to be referenced.

Syntax: pcreate Addr, (VVar, EVar, Addr, GP or WV)

Creates a pointer Addr referencing the given object. This object can be referenced in the rest of the program through this pointer. If the pointer whose name is Addr already exists and references another object, the old reference will be lost. With trace on, the destroyed object will be shown. The object must be valid (VVar, EVar, Addr must exist, Goal Pointer or Working Vertex needs to be not null).

Command: copysg

Parameters:

Addr - > Address of the new pointer, to the copy.

Addr - > Address of the subgraph to be copied.

Syntax: copysg Addr, Addr

Creates a copy of the subgraph started at the vertex whose id is Addr or referenced by Addr. This subgraph can be referenced in the rest of the program through this pointer Addr. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown. The Addr must exist and reference a Vertex.

Command: work

Parameters:

(Addr, VVar or GP) - > Address of the vertex.

Syntax: work (Addr, VVar or GP)

Moves the informed Vertex to the Working Vertex register (WV). The Addr or VVar must exist and reference a Vertex, if the parameter is GP, the Goal Pointer register must be not null.

Command: label

Parameters:

StringVar - > Address of new label.

VVar - > address of the vertex.

Syntax: label StringVar, VVar

Label places in StringVar the content of the vertex whose id is VVar (it must exist). This label can be referenced in the rest of the program through this pointer StringVar. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: source

Parameters:

VVar - > Id that will receive the vertex.

EVar - > Edge's id.

Syntax: source VVar, EVar

Places in VVar the vertex that is the origin of EVar (it must exist). This vertex can be referenced in the rest of the program through this pointer VVAr. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: target

Parameters:

VVar - > Id that will receive the vertex.

EVar - > Edge's id.

Syntax: target VVar, EVar

Places in VVar the vertex that is the destination of EVar (it must exist). This vertex can be referenced in the rest of the program through this pointer VVAr. If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: sedges

Parameters:

SVar - > Set that will receive the edges.

VVar - > Vertex's id or pointer.

Syntax: sedges SVar, VVar

Places in SVar the set of edges whose origin (source edges) is VVar. VVar and SVar must exist.

Command: tedges

Parameters:

SVar - > Set that will receive the edges.

VVar - > Vertex's id or pointer.

Syntax: tedges SVar, VVar

Places in SVar the set of edges whose target (target edges) is VVar. VVar and SVar must exist.

Command: eremove

Parameters:

EVar - > Edge’s id.

Syntax: eremove EVar

Deletes the edge whose id is EVar. EVar must exist.

Command: cmp

Parameters:

(Addr or WV) - > Address.

(String or StringVar) - > Content.

Syntax: cmp (Addr or WV), (String or StringVar)

Compares the content of the vertex, edge, pointer or of the register WV passed with string or StringVar. If equal, sets Compare Register (CR) true, else sets CR false. The Address must exist; else CR will be set as false. If trace is on, null or invalid addresses will be shown.

Command: cmpeorder

Parameters:

EVar - > Edge’s id.

order - >Order integer.

Syntax: cmpeorder EVar, order

Compares the order of the given edge with order integer. If equal, sets Compare Register (CR) true, else sets CR false. The edge must exist; else CR will be set as false. If trace is on, null or invalid addresses will be shown.

Command: cmpedge

Parameters:

EVar - > First edge id.

EVar - > Second edge id.

Syntax: cmpedge EVar, EVar

Compares the order and content of two given edges. If equal, setsCompareRegister (CR) true, else sets CR false. The edges must exist; else CR will be set as false. If trace is on, null or invalid addresses will be shown.

Command: cmpvertex

Parameters:

VVar - > First vertex id.

(VVar or WV) - > Second vertex reference.

Syntax: cmpvertex VVar, (VVar or WV)

Compares the order and content of two given vertices. If equal, sets Compare Register (CR) true, else sets CR false. The vertices must exist; else CR will be set as false. If trace is on, null or invalid addresses will be shown.

Command: sgcmp

Parameters:

VVar - > First subgraph starting vertex id.

(VVar or WV) - > Second subgraph starting vertex reference.

Syntax: sgcmp VVar, (VVar or WV)

Compares the content of two given subgraphs. If equal, sets Compare Register (CR) true, else sets CR false. The starting vertices must exist; else CR will be set as false. If trace is on, null or invalid addresses will be shown.

Command: setCR

Parameters:

(true or false) - > Boolean value.

Syntax: setCR (true or false)

Sets Compare Register (CR) with the Boolean value.

Command: je

Syntax: je

If that CR is true, advances for the next instruction, else it does not advance.

Command: jne

Syntax: jne

If that CR is false, advances for the next instruction, else it does not advance.

Command: mov

Parameters:

(GP, IP, CR or FR) - > First register.

(WV, CR or FR) - > Second register.

Syntax: mov (GP, IP, CR or FR), (WV, CR or FR)

Moves the contents of the second register to the first register. Compare Register (CR) can only be moved to Function Register (FR) and can only receive it also. The remaining combinations of register have no restrictions.

Command: clean

Parameters:

(ALL or GL) - > Register.

Syntax: clear (ALL or GL)

Clear the Goal List (GL) or cleans the machine's internal state (ALL).

Command: gladd

Parameters:

VVar - > Vertex id or pointer.

Syntax: gladd VVar

Adds the referenced vertex to the Goal List (GL). The vertex must exist.

Command: glload

Syntax: glload

Moves the first vertex in Goal List (GL) to Goal Pointer (GP).

Command: lsload

Syntax: lsload

Clear the machine's access points and moves the first vertex in Local Strategy (LS) to it.

Command: mcreate

Parameters:

MVar - > id of match to be created

(String or StringVar) - > content

Syntax: mcreate MVar, (String or StringVar)

Creates the search variable (match) under id MVar and with the content string or StringVar. If MVar already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: match

Parameters:

SVar - > Id of the set.

MVar - > Id of match.

Syntax: match SVar, MVar

Iterates on the set SVar searching for an element whose content is equal to the content of MVar, in case that it finds, it sets MR (MatchRegister) with it, else it sets MR null. MVar and SVar must exist.

Command: jm

Syntax: jm

If MR is true, advances for the next instruction, else it does not advance.

Command: jnm

Syntax: jnm

If MR is false, advances for the next instruction, else it does not advance.

Command: screate

Parameters:

Svar - > Id of the set to be created.

Syntax: screate SVar

Creates a set whose id is SVar. If SVar already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: sadd

Parameters:

SVar - > Set.

Addr - > Object.

Syntax: sadd SVar, Addr

Adds the content of Addr to the Set. SVar and Addr must exist.

Command: sreset

Parameters:

SVar - > Set.

Syntax: sreset SVar

Creates a new iterator over the set SVar (it must exist).

Command: snext

Parameters:

SVar - > Set.

Addr - > Address.

Syntax: snext SVar, Addr

Places in Addr the reference to the next element in the iteration over Svar (it must exist). If the pointer already references another object, the old reference will be lost. With trace on, the destroyed object will be shown.

Command: jse

SVar - > Set.

Syntax: jse SVar

If SVar is empty, advances for the next instruction, else it does not advance. SVar must exist.

Command: jsne

SVar - > Set.

Syntax: jsne SVar

If SVar is not empty, advances for the next instruction, it does not advance. SVar must exist.

Command: push

Parameters:

(IQ, IP, MR, CR, IR, WV ALL or Addr) - > Object.

Syntax: push (IQ, IP, MR, CR, IR, WV ALL or Addr)

Piles up one of the machine's object: IQ - Instruction Queue, IP - Instruction Pointer, MR - Match Register, CR - Compares Register, IR – Interrupt Register, WV – Working Vertex ALL - Current State of the machine or Addr – the object referenced by the pointer.

Command: pop

Syntax: pop

Unpiles one of the objects of the machine, the top object of the stack.

Command: jmp

Syntax: jmp

Unconditional Jump, is equivalent to NOP (no operation).

Command: int

Syntax: int

Interrupts the machine and waits for the decision of an external agent with the program's strategy.

Command: call

Parameters:

Addr - > Vertex id or pointer.

Syntax: call Addr

Make a function call to the vertex. It stores the machine state to be loaded when the instruction ret is executed. The program jumps to the vertex and continues execution there. The vertex must exist.

Command: ret

Syntax: ret

Makes the function return, loading the previous machine state and continuing execution where it made the function call.

Command: print

Parameters:

(LS, IQ, IP, GL or GP) - > Register.

Syntax: print (LS, IQ, IP, GL or GP)

Prints in the output stream the register's name and content.

Command: printsg

Parameters:

(VVar or GP) - > Reference to the starting vertex of the subgraph.

Syntax: printsg (VVar or GP)

Prints the subgraph started on vertex. The vertex must exist.

Command: printsgi

Parameters:

(VVar or GP) - > Reference to the starting vertex of the subgraph.

Syntax: printsgi (VVar or GP)

Prints the subgraph started on vertex ignoring the edges whose content is contained in the set passed in the machine's start. The vertex must exist.

Command: message

Parameters:

String - > message.

Syntax: message String

Print in the output print stream the message parameter.

List of commands for fast reference:

```
vcreate VVar
uvcreate Addr [, String]
ecreate VVar, EVar, VVar
```

```
uecreate (VVar or WV), (VVar or WV), order [, String]
eorder EVar, order
sto Addr, (String or MR)
pcreate Addr, (VVar, EVar, Addr, GP or WV)

copysg Addr, Addr
work (Addr, VVar or GP)
label StringVar, VVar
source VVar, EVar
target VVar, EVar
sedges SVar, (VVar or WV)
tedges SVar, (VVar or WV)
eremove EVar

cmp (Addr or WV), (String or StringVar)
cmpeorder EVar, order
cmpedge EVar, EVar
cmpvertex VVar, (VVar or WV)
sgcmp VVar, (VVAR or WV)
setCR (true or false)
je
jne

mov (GP, IP, CR or FR), (WV, CR or FR)
clean (ALL or GL)
gladd VVar
glload
lsload

mcreate MVar, (String or StringVar)
match SVar, MVar
jm
jnm

screate SVar
sadd SVar, Addr
sreset SVar
snext SVar, Addr
jse SVar
jsne SVar

push (IQ, IP, MR, CR, IR, WV, ALL or Addr)
pop [Addr]
jmp

int
call Addr
ret
print (LS, IQ, IP, GL or GP)
printsg (VVar or GP)
printsgi (VVar or GP)
message String
```

4.4.Inference Model Façade

The machine's interface with its memory is made by the façade shown in Figure 1. In order to understand better the interaction between the machine's processing unit and its memory, the methods signature is copied here. It is a Java [15] interface, but no Java knowledge is needed to understand it. The code relates input and output of each method. These are the methods used by the commands explained in section 4.3.

```
public interface IInferenceModelFacade {
      public static final String TOKEN_ID_SEPARATOR = "#";
      public static final String TOKEN_PARAMETER_SEPARATOR =
",";
      public static final String TOKEN_EXTRA_SEPARATOR =
"&";
      public static final String TOKEN_EDGE_DELIMITER = "$";

      public static final int
EDGE_ORDER_INDICATOR_UNORDERED_EDGE = -1;
      public static final int EDGE_ORDER_INDICATOR_NOT_EDGE
= -2;
      public static final String SPACE = " ";

      public abstract Object getVertex(String id);
      public abstract Object getEdge(String id);
      public abstract Object getObject(String address);

      public abstract void createVertex(String id, String
content);
      public abstract void createUnamedVertex(String
pointer, String content);

      public abstract void createEdge(String id, String
content, int order, String idSource, String idTarget);
      public abstract void createEdge(String id, String
content, int order, Object source, Object target);
      public abstract void createUnamedEdge(String content,
int order, Object source, Object target);

      public abstract void createObject(String address,
Object object);

      public abstract void destroyVertex(String id);
      public abstract void destroyEdge(String id);
      public abstract void destroyObject(String id);

      public abstract void copySubgraph(String copyPointer,
Object vertex, Collection<String> edgeContentIgnore);

      public abstract Object getSource(Object edge);
      public abstract Object getTarget(Object edge);
      public abstract int getOrder(Object edge);

      public abstract Collection<Edge> getSourceEdges(Object
vertex);
```

```
	public abstract Collection<Edge> getTargetEdges(Object 
vertex);

	public abstract String getContent(Object object);

	public abstract String getId(Object object);

	public abstract void setOrder(Object edge, int order);

	public abstract void setContent(Object object, String 
string);

	public abstract void push(Object object);

	public abstract Object pop();

	public abstract void addAccessPoint(String id);

	public abstract void addAccessPoint(Object vertex);

	public abstract void clearAccessPoints();

	public abstract List<Object> 
getNextInstructions(List<Object> queue);

	public abstract void printAll(PrintStream 
printStream);

	public abstract void printMessage(PrintStream 
printStream, String message);

	public abstract void printSubgraph(PrintStream 
printStream, Object object,
			Collection<String> edgeContentIgnore);
}
```