ISBN: 972-8924-09-7 © 2006 IADIS

IMPLEMENTING WATSON-CRICK FINITE AUTOMATA IN *JAVA*[*]

Marcelino Campos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia (Spain)
mcampos@dsic.upv.es

Tomás A. Pérez
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia (Spain)
taperez@dsic.upv.es

José M. Sempere
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n 46022 Valencia (Spain)
jsempere@dsic.upv.es

ABSTRACT

Watson-Crick finite automata (WKFA) were first proposed by Freund et al. in 1997 inspired by formal language theory, finite states machines and some ingredients from DNA computing such as working with molecules as double stranded complementary strings. Here, we propose an implementation of WKFA in Java. The purpose of this wok is to show a complete architecture of methods, classes and interfaces which fully implement a simulator of WKFA with the exception of user interfaces.

KEYWORDS

Computation models, DNA computing, simulators.

1. INTRODUCTION, BASIC CONCEPTS AND NOTATION

Watson-Crick finite automata (WKFA) (Freund et al., 1997) is a good example of how DNA biological properties can be adapted to propose computation models in the framework of DNA computing. The WKFA model works with double strings inspired by double-stranded molecules with a complementary relation between symbols (here, inspired by classical complementary relation between nucleotides A-T and C-G).

In the sequel, we will introduce some basic concepts from formal language theory according to Hopcroft and Ullman ,1979 and Rozenberg and Salomaa, 1997 and from DNA computing according to Păun et al. 1998.

An alphabet Σ is a finite nonempty set of elements named symbols. A string defined over Σ is a finite ordered sequence of symbols from Σ. The infinite set of all the strings defined over Σ will be denoted by Σ^*. Given a string $x \in \Sigma^*$ we will denote its length by $|x|$. The empty string will be denoted by λ and Σ^+ will denote $\Sigma^* - \{\lambda\}$. Given a string x we will denote by x^r the reversal string of x. A language L defined over Σ is a set of strings from Σ.

[*] Work partially supported by the Ministerio de Ciencia y Tecnología under project TIC2003-09319-C03-02.

A grammar is a construct $G = (N, \Sigma, P, S)$ where N and Σ are the alphabets of auxiliary and terminal symbols with $N \cap \Sigma = \emptyset$, $S \in N$ is the *axiom* of the grammar and P is a finite set of productions in the form $\alpha \rightarrow \beta$. The language of the grammar is denoted by $L(G)$ and is the set of terminal strings that can be obtained from S by applying symbol substitutions according to P.

We will say that a grammar $G = (N, \Sigma, P, S)$ is *right linear* (regular) if every production in P is in the form $A \rightarrow uB$ or $A \rightarrow w$ with $A,B \in N$ and $u,w \in \Sigma^*$. The class of languages generated by right linear grammars coincides with the class of regular languages and will be denoted by ***REG***. We will say that a grammar $G = (N, \Sigma, P, S)$ is *linear* if every production in P is in the form $A \rightarrow uBv$ or $A \rightarrow w$ with $A,B \in N$ and $u,v,w \in \Sigma^*$. The class of languages generated by linear grammars will be denoted by ***LIN***. We will say that a grammar $G = (N, \Sigma, P, S)$ is *even linear* if every production in P is in the form $A \rightarrow uBv$ or $A \rightarrow w$ with $A,B \in N$, $u,v,w \in \Sigma^*$ and $|u| = |v|$. The class of languages generated by even linear grammars will be denoted by ***ELIN***. A well known result from formal language theory is the inclusion ***REG*** $\subset$ ***ELIN*** $\subset$ ***LIN***.

A homomorphism h is defined as a mapping $h: \Sigma \rightarrow \Gamma^*$ where Σ and Γ are alphabets. We can extend the definition of homomorphism over strings as $h(\lambda) = \lambda$ and $h(ax) = h(a)\ h(x)$ with $a \in \Sigma$ and $x \in \Sigma^*$. Finally, the homomorphism over a language $L \subseteq \Sigma^*$ is defined as $h(L) = \{ h(x) : x \in L \}$.

Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, we will use the symmetric (and injective) relation of complementarity $\rho \subseteq \Sigma \times \Sigma$. For any string $x \in \Sigma^*$, we will denote by $\rho(x)$ the string obtained by substituting the symbol a in x by the symbol b such that $(a, b) \in \rho$ (remember that ρ is injective) with $\rho(\lambda) = \lambda$.

Given an alphabet Σ, a *sticker* over Σ will be the pair (x, y) such that $x = x_1\ v\ x_2$, $y = y_1\ w\ y_2$ with $x, y \in \Sigma^*$ and $\rho(v) = w$. The sticker (x, y) will be denoted by $\begin{pmatrix} x \\ y \end{pmatrix}$. A sticker $\begin{pmatrix} x \\ y \end{pmatrix}$ will be a complete and complementary *molecule* if $|x| = |y|$ and $\rho(x) = y$. A complementary and complete molecule $\begin{pmatrix} x \\ y \end{pmatrix}$ will be denoted as $\begin{bmatrix} x \\ y \end{bmatrix}$. Obviously, any sticker $\begin{pmatrix} x \\ y \end{pmatrix}$ or molecule $\begin{bmatrix} x \\ y \end{bmatrix}$ can be represented by $x\#y^r$ where $\# \notin \Sigma$. So, inspired by DNA structure, $x\#y^r$ represents the upper and lower nucleotide strings within the same direction 3'–5' (or 5'-3').

Formally, an *arbitrary* WK finite automata is defined by the tuple $M = (V, \rho, Q, s_0, F, \delta)$, where Q and V are disjoint alphabets (states and symbols), s_0 is the initial state, $F \subseteq Q$ is a set of final states and $\delta: Q \times \begin{pmatrix} V^* \\ V^* \end{pmatrix} \rightarrow P(Q)$ (which denotes the power set of Q, that is the set of all possible subsets of Q).

The language of complete and complementary molecules accepted by M will be denoted by the set $L_m(M)$, while the upper strand language accepted by M will be denoted by $L_u(M)$ and defined as the set of strings x such that M, after analyzing the molecule $\begin{bmatrix} x \\ y \end{bmatrix}$ enters into a final state.

Two basic representation results were introduced by Sempere in 2004 as follows

Theorem 1. Let $M = (V, \rho, Q, s_0, F, \delta)$ be an arbitrary WK finite automata. Then there exists a linear language L_1 and an even linear language L_2 such that $L_m(M) = L_1 \cap L_2$.

Corollary 1. Let $M = (V, \rho, Q, s_0, F, \delta)$ be an arbitrary WK finite automata. Then $L_u(M)$ can be expressed as $g(h^{-1}(L_1 \cap L_2) \cap R)$ with L_1 being a linear language, L_2 an even linear language, R a regular language and g and h homomorphisms.

2. AN IMPLEMENTATION IN *JAVA*

In this section we propose an implementation of WKFA in Java. In order to know more about Java we recommend the book by Horstmann and Cornell, 2000.

We have designed a set of interfaces and classes in Java which defines the structures and components needed to develop a WKFA simulator. At the first sight, these software entities are predefined by the structural and operational characteristics of WKFA. So, for example we have classes which are associated to the input tape, to the finite automata, to the complementarity relation, to the instantaneous descriptions of any computation, etc. Some objects of these classes need to interact with others in order to obtain the computation trees. In addition, we need user interfaces to handle the computation trees, the acceptance paths inside the computation trees, etc.

We will describe some classes and methods in this section. Other classes and methods are specified in the appendix at the end of this work.

First, the class **Tape** represents the input data for the automata. It is composed of two equal length strings (*strands*) which are related by a complementarity relation. Anyway, the class **Tape** does not assume any specific complementarity relation. Its definition in Java is described as follows:

```
import java.io.*;

public class Tape implements Serializable{
   //Attributes
   private final String upperStrand;
   private final String lowerStrand;
   //Constructor
   public Tape(String upperStrand,String lowerStrand)
        throws NotValidTapeException{}
   //Methods
   public boolean checkWithComplementarityRelation(
              ComplementarityRelation rel){}
   public int length(){}
   public char getUpperChar(int i){}
   public char getLowerChar(int i){}
   public String getUpperStrand(){}
   public String getLowerStrand(){}
   public boolean equals(Object obj){}
   public int hashCode(){}
   public String toString(){}
}
```

We have omitted the void code for methods and constructors. In the last code, the constructor sends the exception **NotValidTapeException** when the two strings (*strands*) have different lengths. The last three methods are referred to Object. So, for example, the method **public boolean equals(Object obj)** returns *true* when the parameter object is from the class Tape and has identical attributes. These methods have been rewritten for other classes which have identical requirements. The class Tape uses the class **ComplementarityRelation** in the method **public boolean checkWithComplementarityRelation(ComplementarityRelation rel){}** which checks the validity with respect to a given complementarity relation.

Any complementarity relation is defined by the classes **ComplementarityRelation** and **PairOfComplementarySymbols**. The last class defines the complementarity relation between two specific symbols and its code is showed in the appendix of this work. The class **ComplementarityRelation** integrates the set of specific pairs which have been defined by the class **PairOfComplementarySymbols** in order to define a complementarity relation. We show its code as follows

```
import java.io.*;
import java.util.*;

public class ComplementarityRelation implements Serializable{
   //Attributes
   private Set groupOfPairsOfComplementarySymbols = new HashSet();
   //Constructor
   public ComplementarityRelation(){}
   public ComplementarityRelation(String fileName)
        throws IOException,BadDefinedComplementarityRelationException {}
   public ComplementarityRelation(File file)
        throws IOException,BadDefinedComplementarityRelationException {}
   public ComplementarityRelation(InputStream is)
        throws IOException,BadDefinedComplementarityRelationException {}
   //Methods
   public Set getGroupOfPairsOfComplementarySymbols(){}
   public void addPairOfComplementarySymbols(PairOfComplementarySymbols pair)
        throws BadDefinedComplementarityRelationException{}
   public void addComplementarityRelation(ComplementarityRelation relation)
        throws BadDefinedComplementarityRelationException{}
   public void save(String fileName) throws IOException {}
   public void save(File file) throws IOException {}
   public void save(OutputStream os) throws IOException {}
   public boolean contains(PairOfComplementarySymbols pair){}
   public boolean contains(char s1,char s2){}
   public boolean contains(ComplementarityRelation relation){}
   public Set getAlphabet(){}
   public boolean equals(Object obj){}
   public int hashCode(){}
   public String toString(){}
}
public class BadDefinedComplementarityRelationException
                        extends Exception {}
```

Once we have defined WK finite automata, we can handle them by using the following interface. Observe that this strategy separates the construction phase from the running phase.

```
import java.io.*;
import java.util.*;

public interface InterfaceAutomatonWK extends Serializable{
   Set getAlphabet();
   Set getStates();
   Set getFinalStates();
   Set getTransitions();
   Integer getInitialState();
   ComplementarityRelation getComplementarityRelation();
   void setComplementarityRelation(ComplementarityRelation complementarityRelation);
   void save(String fileName) throws IOException;
   void save(File file) throws IOException;
   void save(OutputStream os) throws IOException;
   Set ImmediateTransitions(Tape tape, SnapshotWK snapshot);
   SnapshotWK exeTransition(Tape tape,SnapshotWK snapshot,Transition transition);
   SnapshotWK[] nextSnapshots(Tape tape,SnapshotWK snapshot);
   boolean isFinal(SnapshotWK snapshot,Tape tape);
}
```

The last interface allows the interaction with any object that represents a correctly defined WKFA independently of its implementation. We have assumed the usual simplification of representing every state with an integer. We can observe the existence of the "*get*" methods which allow the recovery of a copy of the main elements of the automata in a normalized format. In addition, two more methods are available to save the automata into a file, so we can use it at any time. This implies that every implementation can read this file to rebuild the automata. This feature will be used in different constructors. Furthermore, they can be used to make an *input stream* persistent. The last three methods allow the interaction between an input tape with the automata through the use of *instantaneous descriptions* (snapshots). So, we can select the admissible transitions and we can execute all of them or everyone in an isolated mode.

The instantaneous descriptions are defined by the class **SnapshotWK** which is showed in the appendix. The objects of this class are pure containers of the elements that define an instantaneous description of a WKFA with an input tape. We have two elements which are defined by two pointers: one of them is linked to the upper strand and the other is linked to the lower strand. They put a mark on the positions of the tape heads and the state of the finite control. In addition, they contain a logical marker, **potentialAcceptance**, which establishes if a final state can be reached from the instantaneous description.

The WKFA have been implemented with the class **AutomatonWK** which is showed again in the appendix.

We have defined **BadDefinedAutomatonWKException** which is an exception that actives itself in the case that the construction or rebuilding of a given WKFA shows any discrepancy between its elements or in its format. In addition, it shows the reasons why the WKFA cannot be correctly constructed. This exception is defined as follows

```
public class BadDefinedAutomatonWKException  extends Exception {}
```

We have different methods to simulate the dynamics of a given WKFA with an input tape. The method **public Set ImmediateTransitions (Tape tape, SnapshotWK snapshot)** allows to obtain the set of transitions of the automata which can be applied to an instantaneous description of an input tape. Obviously, this set can be empty. The method **public SnapshotWK exeTransition(Tape tape, SnapshotWK snapshot, Transition transition)** allows to execute a transition over an input tape for a given instantaneous description. It returns the next instantaneous description if the transition can be applied. Otherwise, it returns *null*. The complete set of the next instantaneous description to a given one can be obtained with the method **SnapshotWK[] nextSnapshots(Tape tape, SnapshotWK snapshot)**. If this set is empty it returns *null*.

The transitions of the WKFA are objects of the class **Transition** which is defined in the appendix. Any object *transition* specifies a basic set which is related to the transitions of the automata. It is constructed by enumerating the starting state, the segment of the upper strand, the segment of the lower strand and the set of arrival states.

Given a WKFA and a (compatible) input tape, the main task of the simulator will consist on obtaining the corresponding computation tree in order to go across its nodes. The basic element to build the computation tree is showed in the class **Node** related to the nodes of the tree. This class is showed in the appendix.

The nodes of the tree are linked in a double way, so we can go from the root to the leaves and in an inverse way. Observe that any object of the class **Node** contains all the references needed to connect it to the tree. This is the significant information of every node inside the computation tree. So, we have linked to every node the instantaneous description together with the transition which is needed to obtain it. In addition, every node has a boolean marker, **potentialAcceptance**, that establishes if we can arrive from this node to an acceptation leaf of the tree.

Once we have constructed a computation tree, we can handle it with the interface **InterfaceComputationTreeWK** which is defined as follows

```
import java.io.*;

public interface InterfaceComputationTreeWK extends Serializable{

   //General methods
   void setAcceptationPaths(InterfaceAcceptationPath acceptationPaths[]);
   InterfaceAutomatonWK getAutomatonWK();
```

```
    ComplementarityRelation getComplementarityRelation();
    Tape getTape();
    boolean checkAcceptance();
    InterfaceAcceptationPath[] getAcceptationPaths();
    void saveTree(String fileName) throws IOException;
    void saveTree(File file) throws IOException;
    void saveTree(OutputStream os) throws IOException;

    //travel methods
    Node getRoot();
    Level getRootLevel();
    Node getNode();
    void setNode(Node node);
    SnapshotWK getSnapshotWK();
    SnapshotWK getSnapshotWK(Node node);
    Transition getTransition();
    boolean exeTransition(Transition t);
    Node exeTransition(Node node,Transition t);
    Node[] getNextNodes();
    Node[] getNextNodes(Node node);
    Node getPreviousNode();
    Node getPreviousNode(Node node);
    int getMinimumDepth();
    int getMaximumDepth();
    Level getDepthLevel(int depth);
    Level getNextLevel(Level level);
    Level getNextLevel();
    int getDepth();
    int getDepth(Node node);
    Node[] getLeaves();
    boolean isLeaf();
    boolean isLeaf(Node node);
}
```

The first methods of the previous code allow to obtain a copy of the initial significant data: the WKFA, the complementarity relation and the input tape. In addition, it is possible to know if the acceptation has been produced by using the method **Boolean checkAcceptance()**. In affirmative case, we can obtain all the acceptation paths by using the method **InterfaceAcceptationPath[] getAcceptationPaths()**. The computation tree can be saved into a file in order to recover it at any moment.

We use the methods **void saveTree(String fileName) throws IOException** and **void saveTree(File file) throws IOException** to make so. We can process the computation tree by using an *InputStream* with the method **void saveTree(OutputStream os) throws IOException**.

The other methods allow to go across the computation tree with different alternatives. Any object of this class has an internal state with the corresponding node (or level) in curse. We can make some operations over these objects, so we can obtain them with the method **Node getNode()**, and put them, if needed, with the method **void setNode(Node node)**. Its instantaneous description can be directly obtained with the method **SnapshotWK getSnapshotWK()** while we can use the method **SnapshotWK getSnapshotWK(Node node)** if we wish to obtain the instantaneous description of any other node inside the tree. The following nodes of the current node of the internal state can be obtained with the method **Node[] getNextNodes()** while the following nodes of any other node can be obtained with the method **Node[] getNextNodes(Node node)**. This working mode is similar to the one defined by the methods **Node getPreviousNode()** and **Node getPreviousNode(Node node)**. The method **Boolean exeTransition(Transition t)** allows to execute the transition of the argument over the node of the internal state. This method, if successful, returns *true* and changes the node by the resulting node obtained from the transition application. If the specified transition cannot be applied then no movement is held and the method returns *false*. In addition, we can use the method

Node exeTransition(Node node, Transition t) which attempts to apply the transition *t* over the node. If it is possible to execute the last transition then the method returns an arrival node, otherwise it returns *null*. The last method does not modify the internal state of the object.

We can work with the levels of the computation tree with the methods **Level getDepthLevel(int depth)**, **Level getNextLevel(Level level)**, **Level getNextLevel()** and **getRootLevel()** which returns the following level of the internal state level or the following level of the internal state if it is the first time that is used. The depths of the nodes can be obtained with the methods **int getDepth()** and **int getDepth(Node node)**. We can directly obtain the leaves of the tree by using the method **Node[] getLeaves()**.

The class **ComputationTreeWK** implements the last interface and must take into account the rebuilding static methods. It returns an object of the class **InterfaceComputationTreeWK** which is useful to rebuild the previously saved computation trees.

The class **Level** which has been previously used is a pure container. It is showed in the appendix.

The interface **InterfaceAcceptationPath** defines an accepting path which is obtained from the computation tree. Its specification is showed in the appendix and we will not make any comment on its methods given that their names are self-explanatory.

Finally, the simulator is defined by the following interface which provides the full functionality of the WKFA

```
public interface InterfaceSimulator{
   void setTape(Tape tape);
   Tape getTape();
   void setAutomatonWk(InterfaceAutomatonWK aut);
   InterfaceAutomatonWK getAutomatonWK();
   InterfaceComputationTreeWK getComputationTreeWK();
   InterfaceAcceptationPath[] getAcceptationPaths();
   boolean checkAcceptation();
   void completeTree();
}
```

The main function of the simulator constructs the computation tree. We show this part as a *pseudocode*

```
AcceptationPaths = empty

LivesNodesList = ComputationTree.root
WHILE LivesNodesList != empty
  Node = getFirst(LivesNodesList)
  IF Nodo is not an acceptance node
    NextNodes = expand(Node)
    FOR ALL Next in NextNodes
      Next.previous = Node
      insert(LivesNodesList,Next)
    END FOR ALL
  ELSE
    setAcceptance(Node,true)
    Path = makeReoute(ComputationTree.root,Node)
    insert(AcceptationPaths,Path)
  END IF
END WHILE
```

3. CONCLUSIONS AND FUTURE WORKS

We have proposed an implementation of WKFA in Java. The internal functions of this automata model and the abilities to handle it have been exposed. Anyway, this is just the first part of a major project involved with

this class of automata. The aspects in which we are working, related to this topic, can be summarized in the following two lines:

- We are developing the user interfaces needed to handle the WKFA model in a friendly manner.
- New variants of WKFA will be implemented. We can mention, among others, *reversal* WKFA, WK *finite transducers*, etc.

Finally, we would like to point out that this tool will be useful in the near future to analyze how these models are able to solve some problems related to biomolecular processing (i.e. DNA/RNA/protein prediction and classification).

REFERENCES

Freund, R. et al., 1999. Watson-Crick Finite Automata. *Proceedings of DNA Based Computers III DIMACS Workshop*, Pennsylvania, USA, pp 297-327.

Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley Publishing Co., Massachusetts, USA.

Horstmann, C.S. and Cornell, G. 2000. *Core Java Vols. 1 and 2*. Prentice Hall, Palo Alto, USA.

Păun, Gh. et al. 1998. *DNA Computing. New computing paradigms*. Springer., Berlin, Germany.

Rozenberg, G and Salomaa, A. (eds.) 1997. *Handbook of Formal Languages Vol*. 1. Springer, Berlin, Germany.

Sempere, J.M. 2004. A Representation Theorem for Languages accepted by Watson-Crick Finite Automata. *Bulletin of the EATCS* No. 83, pp. 187-191.

APPENDIX: METHODS AND CLASSES CODE

```
import java.io.*;

public class PairOfComplementarySymbols implements Serializable{
   //Attributes
   private final char symbol1;
   private final char symbol2;
   //Constructor
   public PairOfComplementarySymbols(char symbol1,char symbol2){}
   //Methods
   public char[] getSymbols(){}
   public boolean equals(Object obj){}
   public int hashCode(){}
}

**************************************************************************

import java.io.*;

public class SnapshotWK implements Serializable{
   //Attributes
   public final int upperStrandPointer;
   public final int lowerStrandPointer;
   public final Integer state;
   private boolean potentialAcceptance;
   //Constructor
```

```
    public SnapshotWK(int upperStrandPointer,int lowerStrandPointer, Integer state){}
    //Methods
    public void setPotentialAcceptance(){}
    public boolean getPotentialAcceptance(){}
    public boolean equals(Object obj){}
    public int hashCode(){}
    public String toString(){}
}

**************************************************************************

import java.util.*;
import java.io.*;

public class AutomatonWK implements InterfaceAutomatonWK{
    //Attributes
    private final Set alphabet;
    private final Set states;
    private final Set finalStates;
    private final Set transitions;
    private ComplementarityRelation complementarityRelation;
    private final Integer initialState;
    //Constructors
    public AutomatonWK(Set alphabet,Set states,Set finalStates,Integer initialState,Set transitions,
                ComplementarityRelation complementarityRelation){}
    public AutomatonWK(Set alphabet,Set states,Set finalStates,Integer initialState,Set transitions){}
    public AutomatonWK(InterfaceAutomatonWK aut){}
    //Methods
    public static InterfaceAutomatonWK rebuilt(String fileName)
            throws IOException,BadDefinedAutomatonWKException {}
    public static InterfaceAutomatonWK rebuilt(File file)
            throws IOException,BadDefinedAutomatonWKException {}
    public static InterfaceAutomatonWK rebuilt(InputStream is)
            throws IOException,BadDefinedAutomatonWKException {}
    public Set getAlphabet(){}
    public Set getStates(){}
    public Set getFinalStates(){}
    public Set getTransitions(){}
    public Integer getInitialState(){}
    public ComplementarityRelation getComplementarityRelation(){}
    public void setComplementarityRelation(ComplementarityRelation complementarityRelation){}
    public void save(String fileName) throws IOException {}
    public void save(File file) throws IOException {}
    public void save(OutputStream os) throws IOException {}
    public Set ImmediateTransitions(Tape tape,SnapshotWK snapshot){}
    public SnapshotWK exeTransition(Tape tape,SnapshotWK snapshot,Transition transition){}
    public SnapshotWK[] nextSnapshots(Tape tape,SnapshotWK snapshot){}
    public boolean isFinal(SnapshotWK snapshot,Tape tape){}
    public int hashCode(){}
    public String toString(){}
}
```

```
**************************************************************************

import java.io.*;
import java.util.*;

public class Transition implements Serializable{
   //Attributes
   public final Integer startState;
   public final String upperStrand;
   public final String lowerStrand;
   public final Integer arrivalState;
   //Constructor
   public Transition(Integer startState,String upperStrand,String lowerStrand,Integer arrivalState){}
   //Methods
   public boolean equals(Object obj){}
   public int hashCode(){}
   public String toString(){}
}

**************************************************************************

import java.io.*;
import java.util.*;

public class Node implements Serializable{
   //Attributes
   private final SnapshotWK snapshot;
   private final Node previousNode;
   private Set nextNodes = new HashSet();
   private boolean potentialAcceptance;
   public final Transition transition;
   private Level level;
   //Constructor
   public Node(SnapshotWK snapshot,Node previousNode,Transition transition){}
   //Methods
   public boolean getPotentialAcceptance(){}
   public void setPotentialAcceptance(){}
   public void  setLevel(Level level){}
   public Level getLevel(){}
   public Node getPreviousNode(){}
   public SnapshotWK getSnapshotWK(){}
   public void addNextNode(Node node){}
   public void addNextNodes(Set nodes){}
   public Set getNextNodes(){}
   public int getDepth(){}
   public int calculateDepth(){}
   public boolean equals(Object obj){}
   public int hashCode(){}
   public String toString(){}
}
```

```
**************************************************************************

import java.io.*;
import java.util.*;

public class Level implements Serializable{
   //Attributes
   private final Node[] nodes;
   public final boolean acceptance;
   public final int depth;
   private Level nextLevel;
   //Constructor
   public Level(Node[] nodes,int depth){}
   //Methods
   public void  setNextLevel(Level level){}
   public Level getNextLevel(){}
   public Node[] getNodes(){}
   public Node[] getAcceptationNodes(){}
   public boolean equals(Object obj){}
   public int hashCode(){}
   public String toString(){}
}

**************************************************************************

import java.io.*;

public interface InterfaceAcceptationPath extends Serializable{
   InterfaceAutomatonWK getAutomatonWK();
   ComplementarityRelation getComplementarityRelation();
   Tape getTape();
   void save(String fileName) throws IOException;
   void save(File file) throws IOException;
   void save(InputStream is) throws IOException;
   void initPath();
   Node getRoot();
   Node getLeaf();
   Node getNextNode();
   Node getNode();
   Transition getTransition();
   Node getPreviousNode();
   SnapshotWK getSnapshotWK();
   SnapshotWK getSnapshotWK(Node node);
   int getMaximumDepth();
   int getDepth();
   int getDepth(Node node);
}
```