Eight-Tuple Description
Linguistic Geometry
Introduction

- The *Eight-Tuple* description has its roots in Linguistic Geometry which has been developed as a generic approach for a certain class of multiagent complex systems.

- Definition: *Complex System* is the following eight-tuple:
  \[<X, P, Rp, SPACE, val, So, St, TR>,\]
This term indicates the space (finite set of points) for the battlefield scenario.

Syntax:
- $X = \{ \text{Domain} [\text{ }, \text{Domain}]^* \}$
- Domain = Start ' ' End ' ' Step
- Start = Number
- End = Number
- Step = Number
- Number = <NUMBER>

Example:

$X: \{1:25:1, 1:25:1\}$
P:

- This term indicates the team of players that are taking part in the game. Here the configuration of the initial players is maintained.

**Syntax**

- \( P \) = \{' Party ['; Party ]* '}
- Party = \{' Element ['; Element ]* '}
- Element = ElementID . Type \{' ResourceType ['; ResourceType ]* '}
- ElementID = Identity
- Identity = <IDENTITY>
- Type = Identity
- ResourceType = Identity

**Example:**

```
P:

{f1.friend(gas, amu, energy), f2.friend(gas, energy),
  e1.enemy(gas, amu), e2.enemy(gas, amu, energy)}
```
Rp

- Reachability can be best defined as the maximum distance move any element can make.

Syntax:
- Rp = ‘{’ Reachable [ ‘,’ Reachable ]* ‘}’ ;
- Reachable = ‘(’ [ ElementID ‘,’ ] Property ‘,’ Relation ‘,’ Condition ‘)’ ;
- Property = Identity ;
- Relation = ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘!=’ ;
- Condition = Number ;

Example:

```lang
Rp:
{(distance, <, 15), (f2, distance, <, 12)};
```

Function provided by the user

Default reachability

Custom Reachability
Val

- Usually this term is the distance which the user wants to calculate.

- Syntax:
  - `Val` = `<JAVA> ':' JavaFile [ ',' JavaFile ]* ;`
  - `JavaFile` = `ClassName '.' MethodName '{' <JAVAFILE> '}' ;`
  - `ClassName` = Identity ;
  - `MethodName` = Identity ;

- Example:

```java
JAVA : ValueFunction.distance @ import java.math.*;
public class ValueFunction {
  public Double Distance(int x1, int y1, int x2, int y2){
    return Math.sqrt(Math.pow(x2-x1,2) + Math.pow(y2-y1,2)); } }
```
Space

- This term is used to indicate the location which are either allowed/denied for certain elements.

- Syntax:
  - SPACE = [ AllowDenyList ];
  - AllowDenyList = ‘{’ ( <ALLOW> | <DENY> ) ‘:’ Position [ , Position ]* ‘}’ ;
  - Position = ‘(’ ElementID ‘,’ Location ‘)’ ;
  - Location = ‘(’ Number [ , Number ]* ‘)’ ;

- Example:

```
SPACE:

{ DENY: (f1,(0,3))};
```

Individual Deny point for the element
This indicates the initial configuration. The current location and the resource allocation can be mentioned here.

Syntax:
- \( S0 = \{ \) Element\_State [ \( ', ' \) Element\_State ]* \( \} \);
- Element\_State = \( ( \) ElementID \( ', ' \) Location \( ', ' \) Resource [ \( ', ' \) Resource ]* \( ) \) \);
- Resource = ResourceType \( : ' \) ResourceQuantity ;
- ResourceQuantity = Number ;

Example

Initial configuration for element "f1"

So:
\( \{ (f1, (1, 2), (\text{gas : 23}, \text{amu : 20})), (f2, (4, 1), (\text{gas : 9})), (e1, (1, 4), (\text{gas : 10}, \text{amu : 35})), (e2, (3, 4), (\text{gas : 10}, \text{amu : 18})) \} ; \)
St

- It indicates the terminating condition. This like the "Val" term is decided by the user.

- Syntax:
  - \( St = \langle JAVA \rangle \; \text{?' JavaFile} ; \)

- Example:

```java
St: Terminating condition provided by the user

JAVA:TerminationState.TerminationCondition @ some java code for termination condition method; @ ;
```
The user decides what kind of move does he want the friendly elements to have if possible.

Syntax:

- TR = <JAVA> '?' JavaFile ;

Example:

```
TR: Transition.TransitionRules @ some java code for transition rules; @
```