WvW/scenario scripting language

Scenario script language

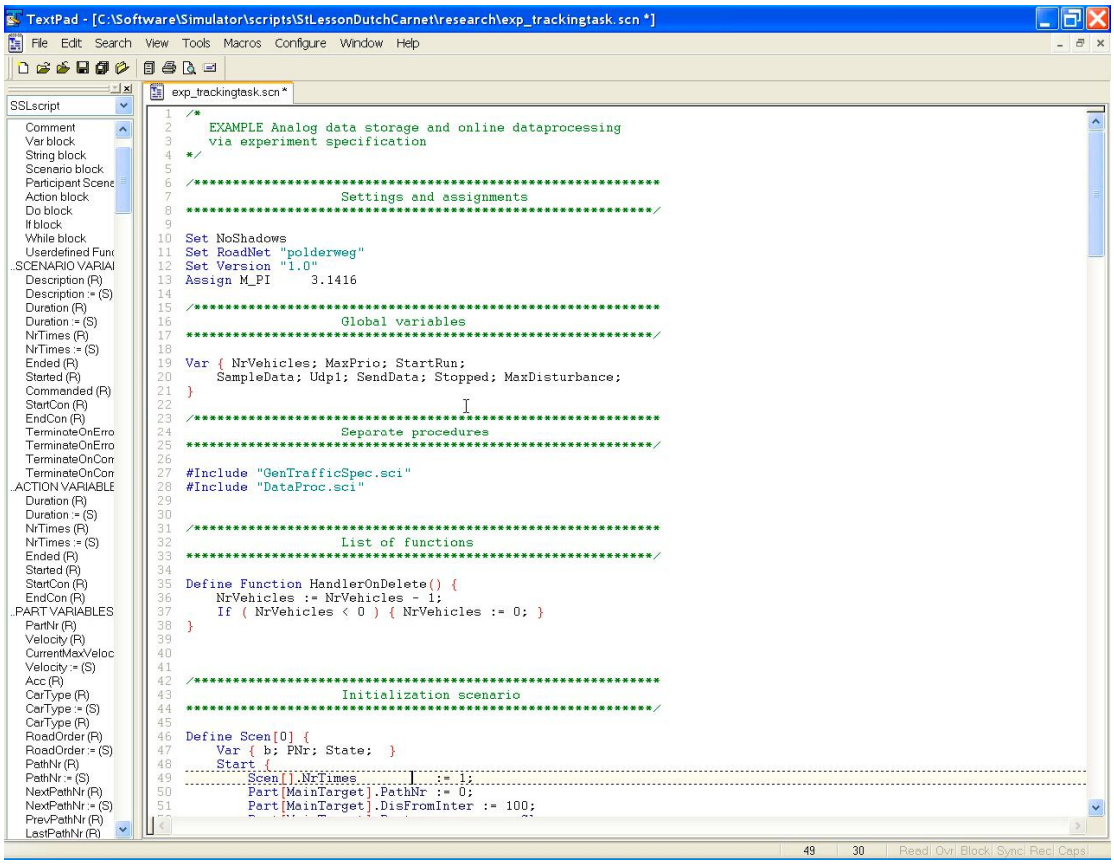
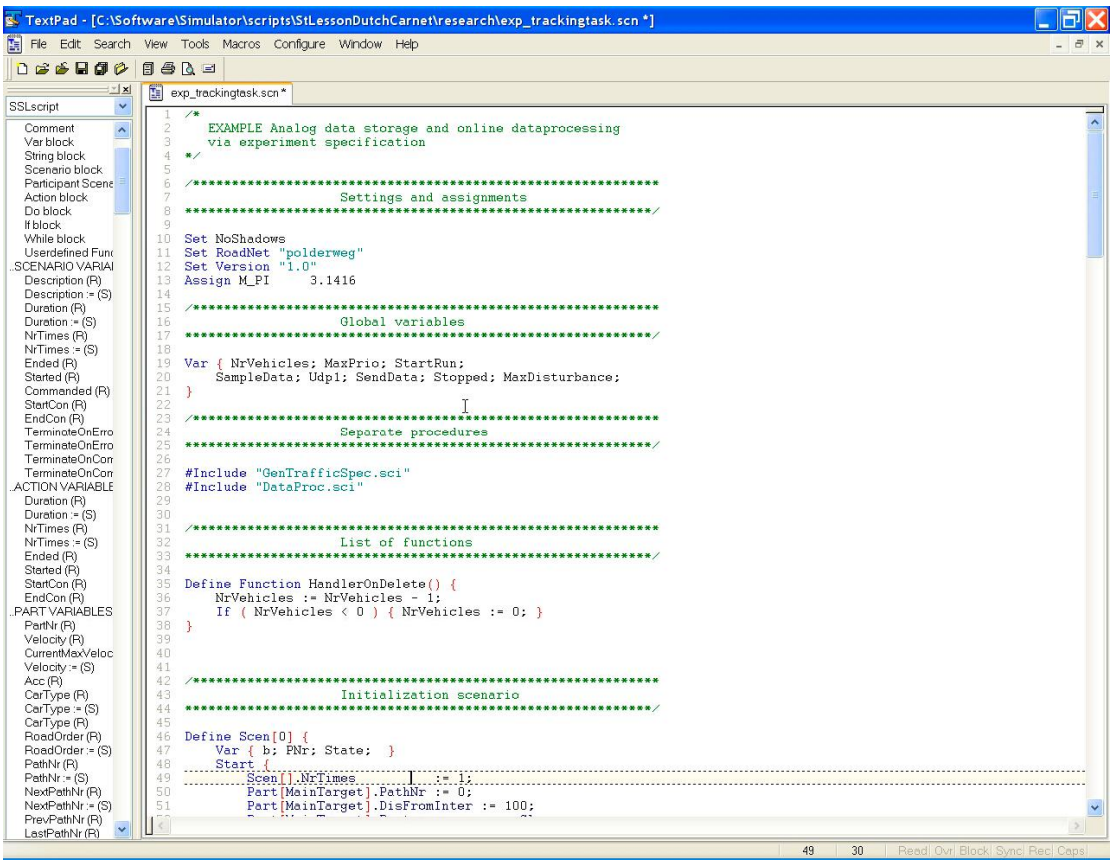


TABLE OF CONTENTS

1 General

1.1 Input and output files

Scenario scipting language is a structured way to specify driving simulator scenarios. It consists of commands in an ASCII file (source file) that are read by traffic.exe. The s*cenario scanner* first performs a syntactical analysis of the specification in the source file. The s*cenario parser* makes a further analysis of the language elements. When no errors occur, internal scenarios are constructed as binary trees that are handled in runtime by the s*cenario interpreter*. When there are syntactical errors, list of errors is printed on the screen, together with a linenumber in which the error occurred, and the simulator program is aborted. Internally, the scanner/parser mechanism constructs a temporary file that consists of a conjunction of the included scriptfiles and the top-level script file. This temporary file has the name 'scentemp##001'. The line numbers in the errorlist refer then to the linenumbers in this file. Textpad 4 is used as a text editor with special syntactical help. The script files have the extention *.scn. They may use include file (extention *.sci) with functionality that's reused. To read the script in the simulator, it has to be compiled into a binary scriptfile with the *.scb extention. In Textpad 4, the syntax is checked via Tools menu -> Sslsyntaxtcheck. If there are syntactical errors you are referred to the file + linenumber where the error is located. If there are no errors, a *.scb file is created in the same folder as the *.scn file. Error checks can only be done from an *.scn file. If there are then errors in an included *.sci file, this is then led to this file + the line number of the error and the type of error.



A scenario is a predefined list of situations with a start- and an end condition. Scenarios are used for the complete simulation process. It may for instance be used for initialization and repositioning of all cars, for controlling traffic lights, for indicating when data must be stored, for communication with the driver via spoken messages and for sending messages to other devices. In our terminology, a scenario is a predefined script that specifies to the runtime system what to do. A database is a separate entity in our terminology and is no part of a scenario, although the scenario makes use of the database.

1.2 Relations between scenarios

Every scenario is unique in the sense that every scenario has a unique identification number. There is no upper limit to the number of scenarios that can be specified. Every scenario must have an unique number. This is an identification number that can be used by other scenarios to refer to.

Scenarios are allowed to overlap, meaning that more than one scenario may be active at the same time. The number of scenarios that may be active at the same time is unlimited, with the restriction that if a particular scenario is active, it cannot be activated again until it is terminated. So if a certain scenario is active it cannot be activated again during the time it is active, but it may be activated again when it is finished. This means that all scenarios that are active at the same time are different and unique. Local scenarios, that are assigned to traffic participants, are a bit different, however. Each participant (autonomous agent) may have the same scenario attached to it, but still all these instantiations are different because they have their own local variables. The same scenario may be activated more than once in the course of runtime.

So, there are two different types of scenarios, global scenarios and local scenarios. A **global scenario** is the normal type: it is defined as

Define Scen[number] {

}
It starts when a certain condition evaluates to true and it stops when another condition evaluates to true. A **local scenario** is always attached to one or more traffic participants. It is defined as:
Define PartScen[number] {

}
Each local instantiation of a PartScen has its own local variables. Local scenarios are used to control things for specific participants.

2 Variable definitions and constants

The user may define two kinds of variables: numbers or strings.

2.1 Numerical variables

These are defined as follows:

Var { ..;..;..; etc }

For example Var { variable1; variable2; etc }

The definition then starts with the keyword Var. This is followed by a set of {} brackets. Within the brackets, each variable is followed by a ‘;’ sign.

There may be any number of these Var blocks defined. If a Var block is defined outside a scenario definition, then it has 'global scope'.’That means that the variable is known anywhere in the scenario definition file, after the point where it has been defined. For example:

```
Var { MeasuredSpeed; }  // from this moment on measured speed is known
...
...
Define Scen[100] {
```

```
        …
        …
        MeasuredSpeed := 0;
        TotalSpeed := 0;  // this results in an error since TotalSpeed has not been defined yet
}

Var { TotalSpeed; Sdsl; }
```

If a variable has been defined within a scenario definition, then it has ‘local scope’: it is only known within the present scenario. Normally variable names must be unique, but it is allowed to have a variablename that has global scope and another variable with the same name that has local scope. For example:

```
Var { MeasuredSpeed; }

Define Scen[100] {
        Var { MeasuredSpeed; }
        MeasuredSpeed := 0;    // the variable with local scope is referred to
        …
}


Define Scen[101] {
        Var { a; b; }
        ….
        MeasureSpeed := 0;  // the variable with global scope is referred to
}
```

A numerical variable must be initialized somewhere in the script with a number, or the result of an expression, for example:

MeasuredSpeed := 0.5;
Or
MeasuredSpeed := (5*rpm())/TotalSpeed;

2.2 String variables

String variables are defined as:

String { ..;..;..; etc }

For example String { Str1; Str2 }
A String block has the same rules as a Var block (global scope or local scope). A string is a series of characters that are enclosed by “” signs, for example :”This is a string”.
A string must be initialized somewhere in the script:

Str1 := “This is a string example”;

2.3 User defined symbolic constants

A user defined symbolic constant is defined as:

Assign SymbolicConstant numberconstant

For example:

Assign TestScenario 1000

After this, the value TestScenario can be used anywhere in the script, but it cannot be changed by the user. For example:

```
Define Scen[TestScenario] {
        …
        …
}
```

2.4 System defined symbolic constants

In addition, some numbers are system defined symbolic constants. These are constants that you can use, but are predefined within the system. You cannot change them and all are reserved keywords. The following system defined symbolic constants are available:

Table 1. Overview of system defined symbolic constants

Keyword	Value	Application
MainTarget	-2	As an objectreference to a participant, for example Part[MainTarget].Velocity
True	1	Boolean value in condition
False	0	Boolean value in condition
On	1	To assess a status
Off	0	To assess a status
Red	-2	Status of traffic light
Yellow	-3	Status of traffic light
Green	-4	Status of traffic light
YellowRed	-7	Status of traffic light
YellowFlash	-5	Status of traffic light
Blank	-6	Status of traffic light
Absent	-1	To test whether an object is present
Normal	-1	Type of intersection
Roundabout	-2	Type of intersection
GiveRow	-1	Right of way regime when coming form a specified path
RowOnLeft	-2	See above
RowOnRight	-3	See above
RowOnBoth	-4	See above
EqualPriority	-5	See above

HaveRow	-6	See above
LeftLane	-1	To position a participant on DLane 1, in Part[MainTarget].Lane := LeftLane;
RightLane	-3	To position a participant on DLane 0, in Part[MainTarget].Lane := RightLane;
RightShoulder	-4	To position a participant on the rightshouder of a highway in Part[MainTarget].Lane := RightShoulder;
DLane	1	LaneType returned by Part[..].LaneType
HardShoulder	6	LaneType returned by Part[..].LaneType
ExitLaneRight	2	LaneType returned by Part[..].LaneType
EntryLaneRight	4	LaneType returned by Part[..].LaneType
ExitLaneLeft	3	LaneType returned by Part[..].LaneType
EntryLaneLeft	5	LaneType returned by Part[..].LaneType
Left	-1	Direction
Right	-2	Direction
Straight	-3	Direction
Clear	-4	Value to clear a route as in Part[..].Route := Clear;
StoreRoute	-5	Value to start a route as in Part[..].StoreRoute;
IndicatorOff	-1	Indicatorstatus, f.i. If (Part[].Indicator = IndicatorOff) {..}
IndicatorLeft	-2	See above
IndicatorRight	-3	See above
IndicatorAlarm	-4	See above
ErrorTerminateScenario	10	Signal in Proc(SignalHandler, ErrorTerminateScenario); Terminate all scenarios that have the TerminateOnError flag set to True
CommandTerminateScenario	11	Signal in Proc(SignalHandler, CommandTerminateScenario); Terminate all scenarios that have the TerminateOnCommand flag set to True
OnDelete	20	Proc(SetHandlerParticipant, OnDelete, participantid, userdefined functionname): if the participant is deleted then apply functionname. OnDelete is detected in the system
OnRouteError	21	Proc(SetHandlerParticipant, OnRouteError, participantid, userdefined functionname): if the participant commits a route error then apply functionname. OnRouteError is detected in the system
OnCollision	22	Proc(SetHandlerParticipant, OnCollision, participantid, userdefined functionname): if the participant collides with another participant then apply functionname. OnRouteError is detected in the system
OnRoad	1	Returnvalue of Part[..].GetPositionOnRoad. OnRoad indicates that the participant is on the road
OffRoadRight	2	Returnvalue of Part[..].GetPositionOnRoad. OffRoadRight indicates that the participant drives to the right of the road
OffRoadLeft	3	Returnvalue of Part[..].GetPositionOnRoad. OffRoadLeft indicates that the participant drives to the left of the road

3 General settings

There is a limited number of special keywords that refer to general settings that apply to the whole script. These settings are specified as:

Set <Keyword> <value>

The following keywords are available:

- **RoadNet**: this specifies the name of the road databases without the extention. If this keyword is read by the scenario parser, both the logical roadnet database with the extention .net is read and the graphical database names are send to the renderers for loading.
  - All *.net files (the logical databases) MUST be in the \SimCarnet\scenegraphs\ folder.
  - All *.bam files (the graphical database + *.ref file optionally) must be in de \models\ folder. The *.bam files refer to other objects that must be stored in the folders under \models\.
- **Version** : this specifies the version identification string that is displayed in some types of userinterfaces (if there's a version control system installed).
- **NoShadows**: shadow generation is switched off for this database in this script.

Example:

Set RoadNet "intersNL"

There must be 1 and only 1 Set RoadNet ".." statement in the scenarioscript. The use of Set Version is optional. The Set statements are best used at the top of the top-level scenarioscript file.

For example:

```
Set RoadNet "intersNL"          // use the 'bibeko' road database, both the logical and the graphical databases
Set Version "v1.1.0 - 1-03-2012"

Var { Stopped;
    StartRun;
    ResetByCommanded;
    SuperFase;
    StartScen10;
    Scen10Done;
    VeelVerkeer;
}

// user defined functions
…
…

// list of scenarios
...
…
```

4 Include scriptfiles

Existing functionality in scripts can be re-used by including these scripts in a top-level scriptfile. It goes like this:

Include "scriptfilename**"**

This line must not be closed by a ';' sign. Include statements can be used within a top-level scriptfile and also within include files. After the 'include' statement, all userdefined functions and global variables as specified in the include file, can be used in the other script files after the inclusion of the respective include file. Because of this, it is strongly recommended to specify the include statements somewhere at the top of the top-level scriptfile.

For example:

```
Set RoadNet "intersNL"
Set Version "v1.1.0 - 1-03-2012"

Var { Stopped;
    StartRun;
    ResetByCommanded;
    SuperFase;
    StartScen10;
    Scen10Done;
    VeelVerkeer;
}

/*************************************************************
            Separate procedures
*************************************************************/

#Include "GenTraffic.sci"
#Include "DA_DrivingTasks.sci"   // assess driving behaviour

/*************************************************************/
```

5 Comment

User comment is specified in either of the following two ways:

- // : all text on the line after the double backward slash is ignored
- /* …. */ : all text between / * and */ is ignored.

If you want to add comment (to improve readability of the script) after some scriptcode on the same line you use the // comment. For example :

```
#Include "DA_DrivingTasks.sci"   // assess driving behaviour
```

If you want to write your comment over more that one line the /*…*/ mechanism can be used. For example :

```
/*************************************************************
            Separate procedures
*************************************************************/
```

6 Scenario definition

Scenarios are defined within a block as:

6.1 Scenario block

The ‘normal’ scenario is a global scenario that is defined as:

Define Scen[*scenario identificationnumber*] {

}

A local scenario is attached to a traffic participant and it is defined as:

Define PartScen[*scenario identificationnumber*] {

}

Both types have the same rules and syntax, so they are treated simply as ‘scenarios’. The only difference is that a PartScen is always attached to a Participant, for example:

```
Define PartScen[21] {
    Start {
        When (…);
        Part[].MaxVelocity := 50/3.6;  // this participant is the participant that uses this scenario
        ….
    }
}
```

and later in the script when a Participant is defined:

```
PNr := CreatePart( 3 );
If ( PNr > 0 ) {
        ….
      Proc( AddScenario, PNr, 21 );  // here scenario 21 is attached to the participant
}
```

So, in this mechanism, the participant uses this PartScen scenario number 21 and it accesses it’s own data by the Part[] object. This is a participant with the default instantiation, indicated by []. And in this case, the default Participant is the participant who uses this scenario. Each PartScen can be attached to any number of participants.

Define and **Scen** and **PartScen** are keywords. Scen is followed by a set of brackets [] that contain the identification number. An identificationnumber is required, and the user must make sure that the number is unique. If scenario numbers are not unique, the scenario parser generates an error and the program is aborted. Scenario identificationnumbers must be positive numbers (from 0..n). There are no restrictions on the ordering of scenario identificationnumbers. So, the following is allowed:

```
Define Scen[0] {

}

Define Scen[10] {

}

Define Scen[8] {

}
```

The scenario identificationnumber may be any of the following:
- a number
- a userdefined symbolic constant. For example:

```
Assign SCENARIOTERMINATEPARSER   9999
....
Define Scen[SCENARIOTERMINATEPARSER] {

}
```

In this example the symbol constant SCENARIOTERMINATEPARSER has been assigned the value 9999. This is used later on to define a scenario with the scenario identificationnumber 9999.

When the program Traffic is aborted, then the scenario 9999 is activated one more time. So Scen[9999] can be used to close things when the program stops, like closing the data for storage etc.

Another special scenario number is 999. When this is activated it is send to the StControl interface to indicated that the current simulation is finished. This activates the <Stop simulation> button on the StControl interface.

A scenario identificationnumber must not be a function or an expression. So the following are examples of illegal scenario specifications:

```
Define Scen[rpm()] {

}
Define Scen[2*TestNum-3] {

}
```

Scenario definitions may contain the following blocks:

Var {…} a list of numerical local variables
String {….} a list of string local variables
Start {…} a specification of a ‘Start’ condition
End {…} a specification of an ‘End’ condition
Do {…} a specification of a lst of statements that have to be executed each cycle
Define Action {…} a sub-scenario

None of these is required, but when these blocks are used, the following rules must be followed:

1) Always specify a Var or/and a String block in the top of the Scen block
2) After this specify the Start block. The Start block specifies when the scenario will be activated. If the Start block is ommitted, then the scenario will start immediately
3) If you use a Do block, it must be specified between the Start and End block.
4) Define Action specifies a sub-scenario. There may be any number of Actions defined within a scenario specification. If you use actions, they must be specified after the End block.

The Var and String blocks have been discussed earlier.

6.2 Start block

A Start block always has the following structure:

Start {
    When (condition);
    <list of statements>
}

The scenario starts to be active if the condition in When (condition); evaluates to True (=1). From that moment on the scenario is active until the End condition (the condition in the End block) becomes true. As soon as the Start condition becomes True, the list of assignments in the Start block is evaluated. This is done only once.

For example:

```
Define Scen[100] {
   Var { Counter; ThisTime; }
   Start {
        When ( Part[MainTarget].PathNr = 50 and Part[MainTarget].DisToInter < 40.5 );
        Counter := 0;
        ThisTime := runtime();
        …
   }
}
```

The scenario starts as soon as the simulator car (Part[MainTarget]) is somewhere in the world on path 50 (a certain road) and less than 40.5 meters from the next intersection. If that conditions has become true, the local variable Counter is set to 0, and the local variable ThisTime is assigned the current time (the system function runtime()).

6.3 End block

An End block always has the following structure:

End {
    When (condition);
    <list of statements>
}

The scenario ceases to be active if the condition in When (condition); evaluates to True (=1). Then, all statements in the End block are evaluated once, and all actions (defined in the list if actions that go with te scenario) are terminated. For example:

```
Define Scen[100] {
   Var { Counter; ThisTime; }
   Start {
        When ( Part[MainTarget].PathNr = 50 and Part[MainTarget].DisToInter < 40.5 );
        Counter := 0;
        ThisTime := runtime();
        …
   }
   End {
        When ( Part[MainTarget].PathNr = 60 and Part[MainTarget].DisFromInter > 25 );
        ThisTime := runtime() – ThisTime;
        Proc( Print, “Scenario 100 has been terminated. Duration of this scenario: ” );
        Proc( Print, num2str( ThisTime, 5, 2 ));
   }
}
```

In this example the scenario starts when the driver has reached path 50 and is less that 40.5 meters to the next intersection. The scenario stays active until the driver reaches path 60 and

is more that 25 meters from the last intersection. From that point on, the scenario is stopped, and a string with the text “Scenario 100 has been terminated” is written to the console screen. Also the total duration of the scenario is computed and written to the console screen.
Proc is a system defined procedure. In this case the procedure ‘Print’ is used, and this has 1 parameter (a string).

6.4 Do block

A Do block always has the following structure:
Do {
<list of statements>
}

For example:

```
Define Scen[100] {
   Var { Counter; ThisTime; AvgSpeed;  }
   Start {
         When ( Part[MainTarget].PathNr = 50 and Part[MainTarget].DisToInter < 40.5 );
         Counter := 0;
         AvgSpeed := 0;
         …
   }
  Do {
         Counter := Counter+1;
         AvgSpeed := AvgSpeed + Part[MainTarget].Velocity;
  }
   End {
         When ( Part[MainTarget].PathNr = 60 and Part[MainTarget].DisFromInter > 25 );
         AvgSpeed := AvgSpeed/Counter;
         Proc( Print, “Average vehicle speed in Scenario 100 : ” );
         Proc( Print, num2str( 3.6*AvgSpeed, 5, 2 ));
   }
}
```

The Do block in this example contains two statements that are executed during each simulation cycle as long as the scenario is active. After termination of the scenario, the average speed is computed and printed on the console screen. Because vehicle speed (Part[MainTarget].Velocity) is measured in m/s, the result is multiplied by 3.6 to obtain the speed in km/h.
All statements in the Do block are repeated each simulation cycle. If the framerate of StTraffic is high, computations such as these may lead to overflow of variables since they may become very high. Also, Do blocks are computationally more expensive than Start or End blocks. Often the same functionality can be obtained by Actions, for example:

```
Define Action {
  Start {
        Counter := Counter+1;
          AvgSpeed := AvgSpeed + Part[MainTarget].Velocity;
  }
 End {
      When ( Action[].Duration >= 0.1 );
 }
}
```

In this action AvgSpeed is processed 10 times per seconds which is quite enough in practice.

6.5 Define Action block

An Action is a sub-scenario: it also has a Start condition, and End condition and possibly a Do block. Actions are used to do special tasks within a scenario. An Action block has the following structure:

Define Action[*Action identification number*] {
    Start {
        When (condition);
        <list of statements>
    }
    Do {
        <list of statements>
    }
    End {
        When (condition);
        <list of statements>
    }
}

The action identification number must be [0..n], and it must be unique within the present scenario specification. If the Start block is ommited, the action starts immediately. Otherwise the action starts when the Start condition (defined in When (condition)) evaluates to True. If the End condition is ommited, the action terminates immediately. Otherwise it terminates when the End condition evaluates to True. If a Do block is defined, all statements within the Do block are executed each simulationcycle as long as the Action is active.

The following example illustrates the use of Actions. If you need to do a number of things as a procedure in a fixed order, Actions come in handy:

```
Define Scen[LOOK_STRAIGHTON] {
  Var { a; State; MyTTI; StopScenario; WaitUntilMessageFinished;
      RightCarId; LastRightCarId; NrAfterControl; }
  Start {
        When ( LookStraightOn = True );
        State := 0;
        WaitUntilMessageFinished := False;
        StopScenario := False;
        MyTTI := 9999;
        LastRightCarId := 9999;
        NrAfterControl := 0;
  }
  Do {
        MyTTI := TTI();
        RightCarId := Part[MainTarget].RightCar;
  }
  End {
        When ( StopScenario = True or LookStraightOn = False );
        LookStraightOn := False;
  }

  Define Action[0] {
    Start {
        When ( State = 0 and MyTTI < 10 );    // start this action is time to intersection < 10 seconds
        a := SendMessage( 30010, SEND_ALWAYS, 0 );  // send a voice message to driver
    }
    End {
        When ( Action[].Duration > 1.0 );              // action terminates after 1 second
        State := 1;
    }
  }

  Define Action[1] {
    Start {
```

```
        When ( State = 1 );
        a := SendMessage( 30020, SEND_ALWAYS, 0 ); // Send a message to the driver
    }
    End {
        When ( Action[].Duration > 1.0 );
        State := 2;
    }
}

Define Action[2] {
    Start {
      When ( State = 2 );
        a := SendMessage( 30031, SEND_ALWAYS, 0 ); // Send a message to the driver
        NrAfterControl := NrAfterControl + 1;
    }
    End {
        When ( Action[].Duration > 1.0 );
        State := 3;
    }
}

Define Action[3] {
    Start {
        When ( State = 3 );
        a := SendMessage( 30020, SEND_ALWAYS, 0 );
    }
    End {
        When ( Action[].Duration > 1.0 );
        State := 4;
    }
}

Define Action[4] {
    Start {
        When ( State = 4 );
        a := SendMessage( 30041, SEND_ALWAYS, 0 );
    }
    End {
        When ( Action[].Duration > 1.0 );
        State := 5;
    }
}


Define Action[5] {
    Start {
        When ( Part[MainTarget].DisToInter < 40 and State = 5 and Part[MainTarget].Velocity < 5.55 and
                NrAfterControl <= 2 );
        State := 2;
    }
    End {
        When ( State = 2 or (Part[MainTarget].DisFromInter > 10 and Part[MainTarget].DisFromInter < 20));
        // jump back to Action 2
    }
}

// After passing the intersection
Define Action[6] {
    Start {
        When ( State = 5 and Part[MainTarget].DisFromInter > 10 and
                Part[MainTarget].DisFromInter < 20 );
        a := SendMessage( 30010, SEND_ALWAYS, 0 );
    }
    End {
        When ( Action[].Duration > 1.0 );
        State := 6;
    }
}

Define Action[7] {
```

```
    Start {
        When ( State = 6 );
        a := SendMessage( 30030, SEND_ALWAYS, 0 );
    }
    End {
        When ( Action[].Duration > 1.0 );
        WaitUntilMessageFinished := True;
        State := 7;
    }
  }

  Define Action[8] {
    Start {
        When ( State = 8 );
    }
    End {
        When ( Action[].Duration > 4.0 );
        StopScenario := True;
    }
  }

  Define Action[9] {
     Var { b; StopAction; MessDuration; }
     Start {
        When ( WaitUntilMessageFinished = True );
        WaitUntilMessageFinished := False;
        If ( LastMessage > 0 ) {
                MessDuration := MessageDuration( LastMessage );
                StopAction := False;
        }
        Else {
                StopAction := True;
       }
     }
     Do {
        If ( LastMessage > 0 ) {
                b := MessageSendTime( LastMessage );
                If ( b > 0 and (runtime() - b) > MessDuration ) {
                        StopAction := True;
                }
        }
     }
     End {
        When ( StopAction = True or Action[].Duration > 15 );
        State := 8;
     }
  }
}
```

7 Userdefined Functions

There are a wide range of system functions that have a returnvalue and 0..n parameters. In addition, users may define their own functions. These are Userdefined functions. They always have the following structure:

Define Function *FuncName(operantlist)* {
    < List of statements>
    ....
    FuncName := expression; (not required)
}

operantlist : operant1, operant2 ..

The user is then free to define a variable number of operants (0..n). These operants are expressions and they may then be variables, constants, other functions (system- or userdefined) and other expressions (for example a/(b+c)). The statement 'FuncName := expression' conforms to the pascal convention. The user is not required to use this statement. It the statement is not included in the function, the return value of the function is zero (0). If the statement is a member of the function then it may be typed anywhere in the function or it may be used several times. The returnvalue of the function is the result that has been assigned to FuncName. Userdefined functions cannot be defined within a scenario and they must be defined only once. Calls to a userdefined function can only be R-values: on the right side of an assignment for example.

In the following example a function is defined that returns 1 or 0 depending on whether a specified amount of time (time_elapsed) is exceeded. It is used in an example that clocks certain events.

```
Define Function StopWatch( time_elapsed, LastTime ) {
   Var { temp; }
   temp := runtime();
   If ( (temp - LastTime) >= time_elapsed ) {
      StopWatch := 1;
   }
   Else { StopWatch := 0; }
}
Define Scen[20] {
  Var { a; OnTimer; BetweenTime; }
  Start {
            When ( StartScen10 = True and
                     Part[MainTarget].PathNr = 380 and
                     Part[MainTarget].Velocity > 2.77 );
            OnTimer          := 0;
            BetweenTime      := 2;
            …
  }
  Do {
            // do something on a timed basis
            a := StopWatch( BetweenTime, OnTimer );
            If ( a = 1 ) {
                     OnTimer          := runtime();
                     BetweenTime      := 3.0+rnd(3.0);
                     …
            }
            …
  }
  End {
          …
  }
}
```

In this example the function StopWatch is a userdefined function. The function runtime() is a system function. If no returnvalue is defined for a userdefined function, you can use it as a procedure to do certain things, for example reset some global variables:

```
Define Function ResetOnRightLane() {
   If ( OvertakingPhase > 0 ) { OvertakingPhase := -1; }
   ThisLaneType  := DLANE;
   ThisLaneIndex := 0;
   PrevLaneType  := DLANE;
   PrevLaneIndex := 0;
}
```

But even in this case you still need to consider such a function as a R-value, as in
a := ResetRightLane();

8 System defined functions

8.1 General overview

A whole set of system defined functions is available to be used in the scripts. System defined functions have a return value and have the following structure:

FunctionName(<list of parameters>);

All system functions are reserved keywords. Most functions return a number and a few return a string. System functions can be used in any expression, for example:

```
Var { TempVar; }
TempVar := sqrt( sqr(a) + sqr(b) );
```

In this example, **sqrt** and **sqr** are systemfunctions.

A number of classes of functions warrant some special attention since they are part of a framework for problemsolving.

8.2 Creation of traffic participants

All traffic in the traffic system is created on-the-fly with the system defined functions CreatePart. The following example shows how you can create a specific type of car:

```
Define Scen[32] {
  Var { a; b; PNr; OnTimer; BetweenTime; CarCount; }
  Start {
     When ( StartScen11 = True and Part[MainTarget].PathNr = 388 );
     OnTimer              := 0;
     BetweenTime          := 2;
     CarCount             := 0;
}
 Do {
     a := StopWatch( BetweenTime, OnTimer );
     If ( a = 1 and CarCount <= 10 ) {
          OnTimer         := runtime();
          BetweenTime     := 3+rnd(3);
          b := 1+rnd(5);
          PNr := CreatePart( b );  // create a car of type b
          If ( PNr > 0 ) {  // if the creation has been succesfull, the value is > 0
                    CarCount := CarCount+1;
                    // now adjust the properties of the car that has been created
                    Part[PNr].RemoveOnDistance := 300;
                    Part[PNr].MaxVelocity := 50/3.6;
                    Part[PNr].Velocity := 50/3.6;
                    Part[PNr].PathNr := 368;
                    Part[PNr].DisToInter := Path[368].Length - 50;
                    Part[PNr].Lane := RightLane;
                    Part[PNr].RuleOvertaking := Off;
                    b := 1+rnd(10);
                    Part[PNr].Rt        := 0.7+(0.1*b);
                    Part[PNr].Route := Clear;
                    Part[PNr].Route := 416;
                    Part[PNr].Route := 410;
                    Part[PNr].Route := StoreRoute;
          }
    }
 }
 End {
     When ( Part[MainTarget].PathNr = 369 or StartScen11 = False );
 }
}
```

In this example, a series of cars is created (no more than 10) during the lifetime of the scenario. The lifetime of the participants (cars) is controlled by the variable *Part[PNr].RemoveOnDistance*. This variable is used by the system to determine at what absolute distance from the simulator car the participant is deleted. If you specify Part[PNr].RemoveOnDistance := 100; then the participant is removed when it is more than 100 meters away from the simulator car.

8.3 Traffic list functions

Because traffic is created by several different scenarios, the control of the lifetime of traffic can be a bit difficult at times. In order to assist in lifetime control a number of traffic list functions can be used. Traffic can be put in different traffic lists and these lists can be deleted all at once. This gives the user the opportunity to let each scenario create and handle its own traffic. In the following example traffic is created and stored in a traffic list. At the end of the scenario all traffic that was created by the scenario is deleted.

```
Define Scen[32] {
  Var { a; b; PNr; OnTimer; BetweenTime; CarCount; }
  Start {
     When ( StartScen11 = True and Part[MainTarget].PathNr = 388 );
     OnTimer              := 0;
```

```
    BetweenTime          := 2;
    CarCount             := 0;
    Scen[].TerminateOnError := True;
    DeleteListNr    := 2;
 }
 Do {
    a := StopWatch( BetweenTime, OnTimer );
    If ( a = 1 and CarCount <= 10 ) {
        OnTimer          := runtime();
        BetweenTime      := 3+rnd(3);
        b := 1+rnd(5);
        PNr := CreatePart( b );  // create a car of type b
        If ( PNr > 0 ) {  // if the creation has been succesfull, the value is > 0
                CarCount := CarCount+1;
                // now adjust the properties of the car that has been created
                Part[PNr].RemoveOnDistance := 1700;
                Part[PNr].MaxVelocity := 50/3.6;
                Part[PNr].Velocity := 50/3.6;
                Part[PNr].PathNr := 368;
                Part[PNr].DisToInter := Path[368].Length - 50;
                Part[PNr].Lane := RightLane;
                Part[PNr].RuleOvertaking := Off;
                b := 1+rnd(10);
                Part[PNr].Rt       := 0.7+(0.1*b);
                Part[PNr].Route := Clear;
                Part[PNr].Route := 416;
                Part[PNr].Route := 410;
                Part[PNr].Route := StoreRoute;
                b := addtolist( DeleteListNr, PNr );
        }
    }
 }
 End {
    When ( Part[MainTarget].PathNr = 369 or StartScen11 = False );
    a := deletelist( DeleteListNr );  // and kill all cars
 }
}
```

In this example, a series of cars is created (no more than 10) during the lifetime of the scenario. These cars are added to a traffic list. If the list DeleteListNr (=2) has not yet been created by this system while you use addtolist(DeleteListNr, PNr), the such a traffic list will be created by the system. At the end of the scenario all cars in the traffic list are deleted, and the traffic list itself is removed. This method may have the disadvantage that cars are deleted while they are still visible to the simulatordriver: in that case the cars suddenly disappear.

8.4 Datacontainer functions

Datacontainer functions help the user in on-line dataprocessing. Any kind of variable can be added to a container and data can be processed on-line. In the following example, data is sampled each second and stored in containers. These are processed afterwards.
In the following example, vehicle speed and lateral position are sampled each second during the lifetime of the scenario, and these data are stored in datacontainers. On completion of the scenario, the average values are computed.

```
Var { SpeedValues, LateralValues; }
Define Function SampleData( SPEED, LAT ) {
    Var { a; }
    a := AddToData( SpeedValues, SPEED );
    a := AddToData( LateralValues, LAT);
}

Define Scen[5000] {
        Var { a; b; AvgSpeed; AvgLatpos; OnTimer; BetweenTime; }
        Start {
```

```
                    …
                    OnTimer         := 0;
                    BetweenTime     := 1.0;
                    Speedvalues     := 1;
                    LateralValues   := 2;
                    ....
        }
        Do {
                a := StopWatch( BetweenTime, OnTimer );
                If ( a = 1 ) {
                        OnTimer     := runtime();
                        b := SampleData( Part[MainTarget].Velocity, Part[MainTarget].LatPos );
                }
        }
        End {
                …
                AvgSpeed :=  MeanData( SpeedValues );
                AvgLatpos := MeanData( LateralValues );
                a := DeleteData( SpeedValues );
                a := DeleteData( LateralValues );
        }
}
```

8.5 UDP ethernet related functions

This category of functions enables the user to send all kinds of data to other computers via UDP or receive any kind of data from other computers. There can be any number of UDP connections opened to other computers. With these functions the user is able to log data on external computers, control external devices, visualize data on external systems etc. Also, the user is able to control simulator functions via script by another computer.
UDP connections to other computers are created by :

a := OpenUdp(ListId, “ethernetaddress”, port);
for example a := OpenUdp(1, “192.168.0.10”, 2001);

From then on, the UDP connection is referenced by the ListId, as in:

```
a := CloseUdp( 1 );
```

Data can be read of written to this Udp port:

```
a := ReadUdp( 1 ) ;
a := WriteUdp( 1 );
```

There are internal read and write buffers of 1024 bytes long that are reserved and controlled by the system. Suppose that you want to send a byte containing a symbol and a short integer containing some value, this can be accomplished by:

```
Var { Symb; OutValue;  Counter; }
a := ClearUdpOut(1);  // clear the output buffer of udp list id 1
Symb := 10; OutValue := 5;
Counter := 0;
a := UdpOutAddByte( 1, Counter, Symb );
Counter := Counter + 1;
a := UdpOutAddShort( 1, Counter, OutValue );
a := WriteUdp( 1 );
```

The following types of data can be added to a databuffer:
- byte (unsigned char)
- short (short integer, 2 bytes)
- long (long integer, 4 bytes)
- float (floating point number, 4 bytes)
- string (character string, any number of bytes)

Suppose that you want to receive a byte and a short from another computer. This can be accomplished as follows:

```
Var { Symb; InValue; Counter; }
Counter := 0;
a := ReadUdp( 1 );
If ( a > 0 ) {
    Symb := UdpInGetByte( 1, Counter );
    Counter := Counter + 1;
    InValue := UdpInGetShort( 1, Counter );
}
```

8.6 Overview of functions

The following Table gives an overview of all systemfunctions that have a number as returnvalue.

Table 2. Overview of system defined functions that return a number

Functionname	Meaning	Input parameters
	MATHEMATICAL FUNCTIONS	
cos	cosine	1: angle in radians
sin	sine	1: angle in radians
tan	tangens	1: angle in radians
log	natural logarithm	1: number
log10	base 10 logarithm	1: number
sqrt	square root	1: number
floor	largest integer not greater than input number : rounding down	1: number
ceil	smallest integer not less than input number: rounding up	1: number
abs	absolute value	1: number
acos	arc cosine	1: number >= -1.0 and <= 1.0
asin	arc sine	1: number >= -1.0 and <= 1.0
atan	arc tangent	1: number
sqr	square	1: number
rnd	random number between 0..input number-1	1: number > 0: rnd(10) gives a random number from 0..9
min	smallest of 2 numbers	2: number, number
max	largest of 2 numbers	2: number, number
SpeedToObject	gives the speed (m/s) such that the input requirements are true	5: current speed, required speed, distance to object, maximum deceleration, headway (in seconds)
lat2ref	Lateral distance with respect to reference track (>0 = left of track, < 0 = right of track)	0
	SIMULATOR CAR RELATED FUNCTIONS	
gear	gear position (0=free gear, 1..5)	0
gearmode	1=5-speed gear; 2 = automatic gear	0
contact	ignition/key position (0=off, 1=on, 2=starter engine)	0
indicator	indicatorposition (IndicatorOff, IndicatorLeft, IndicatorRight, IndicatorAlarm)	0
gas	Gaspedal position (0..100) percentage	0
brake	Brakepedal position (0..100) percentage	0
brakeforce	Brakeforce in Nm	0
handbrake	Handbrake position (0..100) percentage	0
clutchraw	raw clutch position (0..100)	0
clutch	Clutchposition (0..100) percentage	0
steer	Steering wheelangle in radians	0
headlight	headlights (0=off, 1=on, 2= big light)	0
flashlight	Big lights (0=off, 1=on)	0
warnlight	Lights priority vehicle (0=off, 1=on)	0
pdtbutton	Pdt button (for detection response task) pressed	0

	(0=off, 1 = pressed)	
button1	Button1 pressed (0=off, 1 = pressed)	0
button2	Button2 pressed (0=off, 1 = pressed)	0
seatbelt	Seatbelt ((0=off, 1=on)	0
accel	Longitudinal acceleration in m/s^2	0
lataccel	Lateral acceleration in m/s^2	0
rpm	engine rpm (rotations per minute)	0
IsLead	true if participant is in front of me else false	1: participant id
IsRear	true if participant is behind me else false	1: participant id
GetCollisionCar	car is of vehicle that has been involved in collision with simulator car	0
GetNextDir	direction (Left, Right, Straight) after next intersection ranknr	1: 1..5; intersectionranknumber: 1=next intersection, 2 = intersection after that etc.
fuelflow	current fuel consumption in liters/minute	0
fuelused	number of liters fuel used since last ClearFuelCount	0
enginepower	current engine power in Kw	0
horn	claxon pressed (0=off, 1 = on)	0
siren	siren on (0=off, 1 = on)	0
tlc	geometrically accurate tlc of MainTarget (- = toright, + = to left)	0
tlc_1	approximation of tlc (lateral distance/lateral velocity) (- = toright, + = to left)	0
	ANY PARTICIPANT RELATED FUNCTIONS	
RouteOfCar	gives a pathnumber for a participant and a routeindex	2: participant id, route index (zero based)
dhw	distance headway (bumper to bumper along the path) between a participant and another lead participant (in meters)	2: participant id, lead participant id
DisBetween	absolute distance between coordinate positions of two participants (in meters)	2: participant id, participant id
	SYSTEM STATE RELATED	
runtime	time in seconds sinds start of program	0
nrcars	current number of active cars in traffic	0
GetProgramPause	gives true if the Traffic program is in pause mode, else false	0
	SPEECH MESSAGE RELATED	
MessageSendTime	time when message was send to userinterface for playing	1: id number of speech message
MessageDuration	time length of speech message	1: id number of speech message
IsMessagePlaying	true if message is still playing, else false	1: id number of speech message
	DRIVING LANE RELATED	
GetLaneId	unique lane id	3: segmentnr, lanetype, laneindex
LaneTypeLeft	the lane type (DLane, ExitLaneRight etc) of the lane to the left of input lane id	1: lane id
LaneTypeRight	the lane type (DLane, ExitLaneRight etc) of the lane to the right of input lane id	1: lane id
LaneWidth	Lane width in m.	3: segmentid, lanetype, laneindex
GetTrafLightStatus	Gets the traffic light status (Red, Yellow, Green etc) for a specific lane id. If there's no lane-specific traffic light then it checks it the path the lane is in has a traffic light attaced to it, amd it returns the status	1: lane id GetTrafLightStatus(laneid) can be used for both path-connected and lane connected traffic lights. First find the laneid and then apply the function: Id := GetLaneId(Part[].SegmentNr, DLane, 0); Stat := GetTrafLightStatus(Id);
	CREATION OF TRAFFIC PARTICIPANTS	
CreatePart	returns a participant id for a new car of certain type	1: cartype depending on cartypes.conf file
NrCarTypes	returns the number of cartypes defined in the cartypes.conf file	
CreateActor	Returns the id of an animated object, as in Act1 := CreateActor("miabusiness_walking", 490,	4: x, y, height, heading for initial position

	180, 0.1, 270);	
	TRAFFIC LIST FUNCTIONS	
addtolist	adds a participant id to a traffic list. If the list does not exist, a new one is created	2: trafficlist id, participant id, for example, a := addtolist(2, PNr); participant PNr is added to traffic list 2
removefromlist	removes a participant from a list and also from the traffic system as a whole	2: trafficlist id, participant id, for example, a := removefromlist(2, PNr); participant PNr is removed from traffic list 2 and from the traffic system
isempty	True if trafficlist is empty else False	1: trafficlist id
ismemberof	True if participant id is member of this list, else False	2: trafficlist id, participant id
getfirst	returns the first participant id in the traffic list	1: trafficlist id
getnext	returns the next participant id in the traffic list	1: trafficlist id
getlast	returns the last participant id in the traffic list	1: trafficlist id
getprev	returns the previous participant id in the traffic list	1: trafficlist id
deletelist	deletes the complete list and all traffic that is included in the list	1: trafficlist id
numberlist	returns the number of participants in the list	1: trafficlist id
	DATACONTAINER RELATED FUNCTIONS	
AddToData	Adds a number to a datacontainer. If the datacontainer does not exist, a new one is created	2: Datacontainer id, number
DeleteData	Delete a datacontainer	1: Datacontainer id
MeanData	Gives the mean (average) value of all data in the container	1: Datacontainer id
MinimumData	Gives the smallest value of all data in the container	1: Datacontainer id
MaximumData	Gives the largest value of all data in the container	1: Datacontainer id
SumData	Gives the sum of all data in the container	1: Datacontainer id
SdData	Gives the standarddeviation of all data in the container	1: Datacontainer id
NumberData	Gives the number of dataelements (values) in the container	1: Datacontainer id
DataElement	Gives the value of the indexed data element in the container	2: Datacontainer id, index (zero based), f.i. a := DataElement(1, 4); // return the 5th (4+1) element of container 1
SortData	Sort the values in the container from low to high	1: Datacontainer id
	A StringTable is a special type of DataContainer that contains strings instead of numbers	
AddToStringTable	Add a string to a string table (this is a table of strings that can be found by index). If the StringTable does not exist, a new one is created.	2: StringTable id, string
DeleteStringTable	Delete a StringTable	1: StringTable id
NumberStringTable	Gives the number of strings in the StringTable	1: StringTable id
StringTableElement	Gives the value of the indexed string in the StringTable	2: Datacontainer id, index (zero based),
	TYPE CONVERSION	
str2num	converts a string into a number	1: string
	FILE ACCESS	
ReadString	Reads a string from a named file	1: string (filename). Returns a string, or an empty string if end-of-file has been reached
ReadQuotedString	Reads a string between quotes (“….”) from a named file	1: string (filename). Returns a string, or an empty string if end-of-file has been reached
ReadNumber	Reads a number from a named file	1: string (filename). Returns a number, or -1 if end-of-file has been reached or if not a number
	UDP ETHERNET RELATED FUNCTIONS	
OpenUdp	Opens a UDP socket. 0 is failed, else 1.	3: udp list id, ethernetadress string, portid
CloseUdp	Close the UDP connection. 0 is failed, else 1.	1: udp list id
WriteUdp	Write the writebuffer (1024 bytes long at maximum). 0 is failed, else 1.	1: udp list id
ReadUdp	Read the readbuffer (1024 bytes long at maximum).	1: udp list id

	Returns the number of bytes read	
ClearUdpOut	Clears the output buffer	1: udp list id
UdpOutAddByte	Add a byte (unsigned char) of 1 byte on position bufferindex	3: udp list id, bufferindex, value
UdpOutAddShort	Add a short (short integer) of 2 bytes on position bufferindex	3: udp list id, bufferindex, value
UdpOutAddLong	Add a long (long integer) of 4 bytes on position bufferindex	3: udp list id, bufferindex, value
UdpOutAddFloat	Add a float (floating point number) of 4 bytes on position bufferindex	3: udp list id, bufferindex, value
UdpOutAddString	Add a string of characters on position bufferindex. The string is null terminated and strlen(str)+1 bytes are added to the outputbuffer	3: udp list id, bufferindex, value: value must be a string
UdpInGetByte	Get a byte (unsigned char) from the readbuffer starting from position bufferindex	2: udp list id, bufferindex
UdpInGetShort	Get a short (short integer) from the readbuffer starting from position bufferindex (read 2 bytes)	2: udp list id, bufferindex
UdpInGetLong	Get a long (long integer) from the readbuffer starting from position bufferindex (read 4 bytes)	2: udp list id, bufferindex
UdpInGetFloat	Get a float (floating point number) from the readbuffer starting from position bufferindex (read 4 bytes)	2: udp list id, bufferindex
UdpInGetString	Get a string from the readbuffer starting from position bufferindex: this is read until a 0 is found (null terminated string), for example: Str := UdpInGetString(1); length := strlen(Str);	2: udp list id, bufferindex
	COMMUNICATION with Control	
GetByteArrayValue	Gets the value of a byte in a datacommunication buffer of 256 bytes long. Bytes are set by the FillByteArray Procedure	1: index of buffer (0..255)
lookmode	From either buttons that signal the looking direction or facetracker. Gives the values 0..10, each signifying a specific meaning (see gentraffic.sci)	0
ExpDataDefined	Returns 1 if called from an EXP file (experiment specification file)	0

Table 3. Overview of system defined functions that return a string

Functionname	Meaning	Input parameters
strcat	returns a string that appends one string to another	2: string, string
num2str	converts a number into a string	3: number, fieldwidth, number of digits after comma
SubjectIdent	returns the subject identification string from the EXP file (experiment specification file)	0
date	returns a string with date and time information	0
strpart	returns a substring of an inputstring	3: string, index where to start, number of characters
strlen	returns the number of characters	0
DataDef	Returns Data Specification file name	0
programfolder	Name of folder where all python code is located	0

9 System defined procedures

9.1 General overview

System defined procedures handle some predefined task. The general format is

Proc(Procedurename, <list of parameters>);

All Procedure names are reserved keywords. The parameters may be any expression that results in a value. A procedure does not return a value. In the following example, a string is printed to the console.

```
Var { testvalue; }
String { outmessage; }
….
outmessage := strcat( “Test number “, num2str( testvalue, 3, 0 ));
Proc( Print, outmessage );
```

Here are a few examples of categories of procedures.

9.2 Overview of system defined procedures

Table 4. Overview of system defined procedures

Procedure name	Meaning	Input parameters
	CABIN CAR SETTINGS	
SteerTorqueFact	Set a steertorque factor	1: steertorquefactor
BrakeForceFact	Set a brakeforce factor	1: brakeforcefactor
BrakeMax	Set the maximum brakeforce	1: maximum brakeforce (Newton)
GearMode	Set the gear mode	1: 0..2: 0 = 4 gears, 1= 5 gears, 2 = special automatic, 3 = automatic FOR AUTOMATIC GEAR USE 3
SwitchControl	set control to manual (human controls speed and steering), Automatic (automatic control over speed and steering) or semi-automatic (human controls speed and steering is automatic) or speed_automatic (automatic speedcontrol via Part[MainTarget].MaxVelocity and other rules while steering is human controlled)	1: 1..3: 1 = MANUAL, 2= AUTOMATIC, 3 =SEMI_AUTOMATIC, 4=SPEED_AUTOMATIC
SetSpeed	set the speed of the simulator car. With this function you can manipulate the speed of the simulator car	3: speed (in m/s), gear, flag (On/Off)
MaxRollAng	set the maximum roll angle	1: angle in degrees
MaxPitchAngle	set the maximum pitch angle	1: angle in degrees
ResetCabin	Reset the cabin (engine off, speed = 0 etc)	0
SetLimitMaxVelocity	Set maximum velocity if speed is controlled by simulatorcar instead of human	2: maxspeed (in m/s), flag (On/Off)
ClearFuelCount	(re)sets the fuel counter to zero	0
BrakeFactor	sets the extent to which the car brakes as a function of brakepedal position)	1: Default = 18, but a comfortable value is 7.27
FrictionFactor	sets the roadfriction	1: Default = 0.85. More is larger roadfriction.
SpeedBump	Gives a pulse on the steering wheel if you pass a speedbump	0
SetSteeringDelay	Add a time before the steering wheel responds (for example alcohol simulation)	1: steering delay in secondes
SetBrakeDelay	Add a time before the brake pedal responds (for example alcohol simulation)	1: brake delay in seconds

	STEERING DISTURBANCE	
CrossWind	Sets the static velocity of the crosswind acting upon the vehicle model. The wind is set perpendicular to the driving direction (pos wind is 90 degrees from the left	1: velocity
AlongWind	Sets the static velocity of the longitudinal wind acting upon the vehicle model. The wind is set in the driving direction (pos. wind is opposite wind, neg wind from behind).	1: velocity
RoadBank	Sets the virtual slope in degrees. A positive value represents a banking angle to the right and requires left-steer compensation	1: degrees
RoadSlope	Sets the virtual road slope (not graphically !), positive values represent increased raad slope and will slow down the car, while a negative value increases driving velocity	1: degrees
SteerOffset	Sets the value of the steering wheel offset angle in degrees. A positive value should be compensated for by steering right and vice versa.	1: degrees

	DATA STORAGE PROCEDURES	
OpenData	Open a binary datafile and start datasampling	2: string (binary filename without extention), string (string to include in header of datafile)
CloseData	Stop datasampling and close binary datafile	0
ClearDataVariables	Clear all datavariables for storage in binary file	0
AddDataFunction	Add a UserDefinedFunction for datastorage	1: Name of Userdefined function, between “” characters, as in Proc(AddDataFunction, “MyFunction”);
AddDataVariable	Add a datavariable	1: variable type
SetSampleFrequency	Set the sample frequency for datasampling	1: frequency (Hz)
SetEventCode	Set an eventcode (together with the currenttime) in the eventfile	1: number (eventcode)
SetTimeAndEventCode	Set an eventcode and a time	2: number (eventcode), number (timevalue)
	ROAD NETWORK RELATED	
DefaultMaxSpeed	Set the default maximum velocity for the entire roadnetwork	1: speed (in m/s)
SetCountry	country (used in handling of behavioural rules, these are different for different countries)	1: 0=Netherlands, 1 = Germany etc.
SetTrafLightStatus	Sets the traffic light status (Red, Yellow, Green, YellowRed, YellowFlash, Blank) of a traffic light for a specific lane. If there is no trafficlight then it checks if the path on which the lane is located, has a traffic light. If no traffic light can be found it does nothing.	2: laneid, status (Red, Green, etc). Proc(SetTrafLightStatus, laneid, status) can be used for both path-connected and lane connected traffic lights. First find the laneid and then apply the function: Id := GetLaneId(Part[].SegmentNr, DLane, 0); Proc(SetTrafLightStatus, Id, Red);
	PARTICIPANT RELATED	
DeletePart	Delete a traffic participant	1: participant id
RemoveActor	Delete an actor (animated object) for example Proc(RemoveActor, Act1);	1: actor id
Perform	add a UserDefined function to a participant: the participant evaluates this function each cycle f.i Proc(Perform, 2, “EvaluateThis”);	2: participant id, string (name of userdefined function: this function must have no parameters !)
RemovePerform	remove the UserDefine function for the participant	1: participant id
AddScenario	adds a local scenario (defined by Define PartScen) to the participant	2: participant id, scenarionumber
RemoveScenario	removes a local scenario from a participant	2: participant id, scenarionumber
SetHandlerParticipant	set a userdefined function as signal handler for	3: signal (OnDelete, OnRouteError,

	the participant, f.i. Proc(SetHandlerParticipant, OnCollision, 0, "HandlerOnCollision"); (0 = Participant 0 which is the simulatorcar)	OnCollision), participant id, string (name of userdefined function)
LefthandDriving	Sets the system (all driving rules and participant behaviour) to the British lefthand driving system	1: flag (True, False)
AddRuleSpeed	add a speed value (each cycle) that is integrated in the behavioural rules of the participant	2: participant id, speed (in m/s)
AddRuleLatpos	add a required lateral position	4: participant id, time within it must be reached, goal lateral position, priority
	MAINTARGET RELATED (Simulatorcar)	
RepositionRouteByIndex	Reposition the routepointer according to a route index	1: routeindex (0=first path of route)
SetRouteHandlingSSL	flag to indicate whether the system should reset the route if the car deviates from the planned route	1: flag (True, False)
	TRACK RELATED (tracks for actors)	
ClearTrackActor	Clear all trackelements for the actor id	1: actor id
AddStraightTrackActor	Add a straight track to the tracklist	2: actor id, length of track (in m)
AddCurveTrackActor	Add a curve track to the tracklist	4: actor id, direction (Left/Right), radius (in m), turnangle (in degrees)
StoreTrackActor	trackdefinition is completed: initialize track pointer	1: actor id
StartAnimation	Start a named animation for the actor, f.i.. Proc(StartAnimation, id, "Walk");	2: actor id, string of names animation
SetMaxVelocityActor	Forward speed at which animated object moves.	2: actor id, speed in m/s
	SCENARIO RELATED	
StartScen	Start a scenario immediately, f.i. Proc(StartScen, 10);	1: scenario id
EndScen	terminate a scenario immediately	1: scenario id
SignalHandler	terminate all scenarios with signalhandlerflags set, f.i. Proc(SignalHandler, Command TerminateScenario); this will terminate all scenarios for which the flag TerminateOnCommand has been set to true	1: signalhandlerflag (ErrorTerminate Scenario or CommandTerminate Scenario)
	FILE ACCESS	
OpenFile or OpenFileWrite	Create an ascii file (for writing) to store text data, f.i. Proc(OpenFile, “TempStore”); The can be opened any number of files simultaneously Datafiles MUST be in the folder /data under the script file folder	1: string (filename)
OpenFileRead	Open a existing ascii file for reading data. Datafiles MUST be in the folder /data under the script file folder	1: string (filename)
WriteFile	Write text data to a named file, f.i. Proc(WriteFile, “TempStore”, “sample text”);	1: string (filename), string (text data)
CloseFile	Close the file with the specified name	1: string (filename)
Exec	Execute a dis command, for example Proc(Exec, "ImpairConv.bat");	1: string
	MESSAGES	
Print	Print a string to the console	1: string
ClearMessages	Clear all scheduled speech messages	0
SetMaxPriority	set the maximum priority of scheduled speech messages to be sent to the GUI	1: number (0..3)
ScheduleIsdMessage	send a speech message to the scheduler that will handle and send it to the GUI. The speechmessage id is a number that conforms to a .wav file on the GUI (f.i. Proc(ScheduleIsdMessage, 10023, 1, 0); will play the wav file 10023.wav on the GUI PC.	3: number (speech message id), priority (0..3), flag (always set on 0)
PrintGui	print a message to the blue message box of the StControl userinterface	1: string (between “” characters)
SetDebugFlag	Set the debuglevel for console prints	1: 0 = All Starts and Ends of scenarios are printed to the console. Should be 1 by

		default
	SENDING DATA OVER ETHERNET	
FillByteArray	fill a position in an indexed array with an unsigned character	2: index (0..255), value (unsigned char)
SendByteArray	send the array with 255 bytes via TCP	0
SendAutomationData	Automation data to control for storing in student assessment system	3
	GRAPHICS RENDERING	
StimPicture	Proc(StimPicture, "AU_VoorrangAlgemeen", MainPopupX, MainPopupY, 20, 20, 1);	6: 1) string (bitmap name, must be a *.png file), 2) X pos angle with respect to center of screen, 3) Y pos angle with respect to center of screen, 4) width (angle), 5) height (angle), 6) ON (0 = off, 1 = on,)
SetFog	Set fog intensity and colour	2: 1) fog intensity, 2) fog colour (gray value from 0..1
SetSky	Set a skydome id	1: skydome id
SetDayLight	Either day or nigh driving	2: 1) 1=night, 0 = day. 2) not used
SetBlur	Blurs the image of rendering (as in alcohol simulation)	1: > 1.0 = normal, 0 = maximum blur
SetWeather	Proc(SetWeather, 1, 1);	2: 1) 1=rain, 2= snow particle animations 2) intensity (0..10), not used
ScreenText	Text on rendering display	5: 1) output string, 2) X, 3) Y, 4) 1 = on, 0 = off, 5) display: 0=middle, 1=left, 2 = right
Pdt	Draw PDT red block on display	1: angle, -1 if remove
	VEHICLE MODE	
SetEngineMu	Set road friction	1:
SetEngineMaxPower	Set power of engine	1:
SetEngineRedRpm	Set rpm level cutoff	1:
SetSteeringRatio	Set steering ratio (normally approx. 19)	1:
SetRoll	Enable roll of simulated cabin	1: on/off
SetPitch	Enable pitch of simulated cabin	1: on/off

10 Statements, conditions and expressions

10.1 Statements

The general format of a statement is:

```
 [ident ":=" expression";" |
 "When" condition";"      |
  Procedure ";"           |
 "If" condition { [Statement;...] }
  ["ElseIf" condition { [Statement;...] } ]
  ["Else" { [Statement;...]] |
  'While block']
```

With a statement there is a difference between an assignment (using :=), a condition (in combination with the reserved word When), a procedure, defined as the keyword Proc(procedurename, list of parameters) and If blocks.

Assignment

An assignment is defined as

ident ":=" expression ";"

ident is a so-called left-value. It receives the value of the expression.

When (condition)

This is a statement, terminated with a ";". It is defined as :

"When" (condition) ";".

This statement is used as a start or end condition for scenarios and for actions. They may be absent in which case the start or end condition is assumed to be TRUE.

If block

An if block executes statements depending on a condition. It is defined as

"If" (condition) "{" [statement; [statement;...]] "}"
["ElseIf" (condition) "{" [statement; [statement;...]] "}"]
["Else" "{" [statement; [statement;...]] "}"]

An If and an ElseIf must be followwed by a condition. An Else must not be followed by a condition. An ElseIf or an Else must be preceded by an If. In an If block 0..n ElseIf are allowed, but only 1 Else if allowed. This Else must be the last subblock of the If block.

While block

A while block repeatedly executes statements while a condition is True. Within the block the condition must become False. This repetition takes place within 1 simulation frame.

"While" (condition) "{" [statement; [statement;...]] "}"

10.2 Condition

A condition has the general format:

[expression ("=" | "!=" | "<" | "<=" | ">" | ">=") expression |
 expression ("and" | "or") expression]

The logical operators {and, or} can be used. Furthermore, there are the following relational operators: =, != (not equal to), <, <=, > and >=.

An elementary condition has the operators =, !=, <, <=, > or >= between two expressions. A compound condition consists of elementary conditions, separated by the operators "and", "or".

A condition must be preceded with the word "When", or with the keyword "If" or "ElseIf". Because a "When condition" clause is a statement it must be closed with a ";". "If", "ElseIf" and "Else" clauses are blocked. This means the block always starts with a "{" and ends with a "}", but the "}" is never terminated with a ";".
Priorities of elementary or compound conditions can be expressed by extra brackets, for example :

When ((a and b) or (c and d));

10.3 Expression

An expression has the general format:

["+" | "-"] term {("+" | "-") term}

Term

factor {("*" | "/") factor}

Factor

ident | number | "(" expression ")" | Function

Function

A function returns a value, has a name and 0..n parameters, separated by commas. The parameters are expression. It can be either a Userdefined function or a system define function.

functionname "(" [expression ["," expression ...]] ")"

For example : tan(var2*abs(33*var1/var3))
has 1 parameter. It returns the tangent of the input expression.

Ident

object "[" [ident | number | constant | UserDefinedVariable] "]" "." variable
 || UserDefinedVariable

For the object type see chapter 11.

11 Objects

11.1 General overview

The system knows a number of different types of objects. The names of objects are reserved keywords:

- **Scen, PartScen** (a scenario)
- **Part** (a participant)
- **Action**
- **Inter** (an intersection)
- **Path**
- **Segment**

An object is a general data type that is instantiated by an ident, a number or a symbolic constant between []. For example Scen[1] or Part[3]. In this case a number is used. Also, a symbolic constant with a logical meaning can be used, like Part[MainTarget]. A userdefined symboloc constant may also be used, as in

```
Assign DETECTSPEED        2000
…
a := Scen[DETECTSPEED].NrTimes;
```

When there is no instantiation given, the default object instantiation is used. Also an ident may be used to express the instantiation, for example :

```
Path[Path[Part[MainTarget].PathNr].PathFromRight].Length
```
or
`Path[Var_1].Length`, where Var_1 is a UserDefinedVariable that has been initialized before.

With a certain objecttype, a set of variables may be used. For example, Part[].Velocity is the speed (in m/s) of the current (default) participant.
The variables can be changed with an assignment, f.i. Part[].Velocity := 10; or read, f.i. :
Part[].Velocity := Part[MainTarget].Velocity; or :
When (Part[MainTarget].Velocity <= 20);

When an ident is left of the assignment (":=") symbol, the ident is an l-value. In all other cases it is an r-value. An r-value is being "read" and an l-value is being "set". With nested identifiers, all inner identifiers (identifiers within the first [..]) are r-values. With an assignment, only the outer identifier is an l-value. Some variables belonging with a certain object are only allowed to be r-values.

11.2 Scen object, PartScen object

A scen object is a scenario. Variables can be set or read. A PartScen is a scenario that is allocated to a specific Participant. The difference between these two types of scenarios are:
- a Scenario is a global scenario. Only 1 instantiation can be active at the same time. So, if Scen[4] is active it must stop before it can be active again.
- a PartScen is a local scenario and the same local scenario can be attached to any number of participants. This is done via the following Procedure:
Proc(AddScenario, PNr, PartScen number); This mechanisms is used to let a specific car perform a certain task.

The following table gives an overview of variables for this object type. An example to set a variable:

```
Scen[100].NrTimes := 2;
```

You can also read a viariable as in :

```
Define Scen[101] {
    Start {
            ….
          Scen[].NrTimes := Scen[100].NrTimes;
    }
}
```

Table 5. Variables of objecttype Scen

Variable	Read	Set	Value
Description	+	+	String
Duration	+	+	number
NrTimes	+	+	number
Ended	+	-	flag (True/False)
Started	+	-	flag (True/False)
Commanded	+	-	flag (True/False)
StartCon	+	-	flag(True/False)
EndCon	+	-	flag (True/False)
TerminateOnError	+	+	flag (True/False)
TerminateOnCommand	+	+	flag (True/False)
Type	+	-	number (0=global scenario, 1=local scenario PartScen)

Description is a string ("...") that is used in the GUI as a scenario description. It can not be changed during runtime of the simulator. If the variable Description has been set, the descriptionstring will be send to the GUI and displayed in the list of scenarios. The operator is then able to monitor which scenarios are current and which have been activated.

Duration specifies the time duration of the scenario. Default this is infinite. When Duration is set, the maximum time duration is specified. When no other endcondition becomes true, the scenario is aborted after this time. When Duration is being read, the time since the moment the scenario was activated is given.

NrTimes specifies the number of times the scenario may be activated. When this is set, the maximum number of times the scenario may be activated is specified. When it is read, the number of times the scenario was activated (since runtime, inclusive of the current activation) is given.

Ended is a flag that indicates whether the scenario is ended. It is True after the scenario is finished.

Started is a flag that indicates whether the scenario is active. It can only be read.

Commanded is a flag that indicates whether the scenario may be called from outside the program (via the GUI). It can only be read. It may be applied as:

```
Start {
  When ( Scen[].Commanded = True );
}
```

In that case the scenario will start if it has been selected on the GUI (via select scenario and start scenario). In order to be shown on the interface, also the variable ‘Description’ must be set.

StartCon can only be read and returns True or False depending on whether the startcondition for scenario activation is still True or False. This can for instance be used in the endcondition to switch the scenario off if the startcondition is no longer True, thus simulating a "while loop". It can also be used for simulation of "if .. then else if ... then etc."

constructions. For example:

```
Define Scen[100]  {
   Start {
         When ( Part[MainTarget].Velocity > 33.3 );
         ….
   }
   End {
          When ( Scen[].StartCon = False );
   }
}
```

EndCon can only be read and returns True or False depending on whether the endcondition of the scenario is True or False.

TerminateOnError is a flag that indicates whether the scenario must terminate when an errorcondition occurs. The error condition is signalled via Proc(SignalHandler, ErrorTerminateScenario);

TerminateOnCommand is a flag that indicates whether the scenario must terminate when a scenario is switched on via the 'Commanded' variable. This is signalled via Proc(SignalHandler, CommandTerminateScenario); In the following example a scenario is specified that is activated from the userinterface (GUI). If it is commanded to start, it's own flag 'TerminateOnCommand' is set to false, to avoid that it is killed immediately by the signalhandler. Then the signalhandler is called with the instruction to terminate any other scenario that was started from the GUI (i.e., scenarios that have the Commanded flag set to true). After that, the flag TerminateOnCommand is set to true, so that this scenario is terminated if another scenario is commanded to start from the GUI.

```
Define Scen[74] {
   Var { a; }
   Start {
     When ( Scen[].Commanded = True or StartScen20 = True );
     Scen[].Description := "Merging into traffic;
     If ( Scen[].Commanded = True ) {
          Scen[].TerminateOnCommand := False;
          Proc( SignalHandler, CommandTerminateScenario );
          SuperFase := 0;
          StartScen20 := True;
     }
     Scen[].TerminateOnCommand := True;
   }
   End {
     When ( StartScen20 = False );
   }
}
```

11.3 Action object

An Action object is an action. Variables can only be read or set within the scope of the scenario for which they apply.

Table 6. Variables of objecttype Action

Variable	Read	Set	Value
Duration	+	+	number
NrTimes	+	+	number
Ended	+	-	flag (True/False)
Started	+	-	flag (True/False)
StartCon	+	-	flag(True/False)
EndCon	+	-	flag (True/False)

The meaning of these variables is comparable to that of the Scen object.

11.4 Inter object (intersection)

An intersection is a node where more than one roads connect. Variables of this object type can only be read because an intersection is part of the static roadnet structure that cannot be changed on-line.

Table 7. Variables of objecttype Inter

Variable	Read	Set	Value
NrArms	+	-	number
Controlled	+	-	flag (True/False)
NodeType	+	-	intersectiontype: 0..3; 0 = Normal_Int, 1 = Deadend_Int, 2 = Virtual_Int, 3 = Roundabout_Int

NrArms gives the number of branches of an intersection. A T-junction has three branches and an X-crossing has 4 branches.

Controlled is a flag that indicates whether the intersection is controlled by trafficlights. In that case Controlled = True.

NodeType returns the type of intersection. Type can be 0 (normal intersection), 1 (Deadend intersection: the road stops after this road, intersection has only 1 branch) , 2 (Virtual intersection: not a real intersection but a connection between two roads, intersection has 2 branches), 3 (roundabout, intersection is part of a roundabout complex).

11.5 Segment object

A segment is a part of a road. It can be either straight or curved. Variables of this objecttype can only be read because a segment is part of the static roadnet structure. A segment has 0..n lanes. These lanes are of a specific type. DLanes are normal driving lanes. These should be at least 1 DLane on each segment. An ExitLaneRight, ExitLaneLeft, EntryLaneRight, EntryLaneLeft, HardShoulder, HardShoulderLeft, BicycleLaneRight, BicycleLaneLeft, PavementRight and PavementLeft are special lanetypes.

Table 8. Variables of objecttype Segment

Variable	Read	Set	Value
Length	+	-	number (in meters)
Radius	+	-	number (in meters)
NrDLanes	+	-	number
NrExitLanesRight	+	-	number
NrExitLanesLeft	+	-	number
NrEntryLanesRight	+	-	number
NrEntryLanesLeft	+	-	number
NrHardShoulders	+	-	number
Width	+	-	number

Length is the length of the segment in meters, measured along the centerline of DLane[0] (the rightmost driving lane).
Radius is the radius of the segment in meters, from the centerpoint to the middle line of DLane[0]. If the segment is straight, the radius = 0.

NrDLanes is the number of DLanes.
NrExitLanesRight is the number of lanes of type ExitLaneRight.
NrExitLanesLeft is the number of lanes of type ExitLaneLeft.
NrEntryLanesRight is the number of lanes of type EntryLaneRight.
NrEntryLanesLeft is the number of lanes of type EntryLaneLeft.
NrHardShoulders is the number of lanes of type HardShoulder (rightmost lane on highway, 0 or 1).

Width is the roadwidth in meters.

11.6 Path object

A path is a logical connection between two intersections, or between two connectionnodes of between an intersection and a connectionnode. It has a direction from Node A to Node B (with Node being an intersection or a connectionnode). Most variables of this object type can only be read. An exception is the status of a trafficlight group at the end of the path. This can be also be set. Often there are more than one trafficlights at the intersection at the end of a path. Setting the trafficlight sets all trafic lights at the end of the path and reinitializes all trafficlights that are part of the trafficlight group. Also the variable EntranceAllowed can be set. Although this variable is set during roadnetwork initialization (depending on signs), the setting can be overruled, for example if you want a certain road to be a one-way street. In a similar manner is it allowed to change the rigt-of-way regime at an intersection (at the end of the path from the direction of the path).

Table 9. Variables of objecttype Path

Variable	Read	Set	Value
Length	+	-	number (in meters)
NrSegments	+	-	number
TrafficLight	+	+	number (Red, Yellow, Green, YellowRed, YellowFlash, Blank. If there is no trafficlight then the

			value is Absent)
GreenPhase	+	+	number (seconds)
YellowPhase	+	+	number (seconds)
YellowRedPhase	+	+	number (seconds)
PathFromRight	+	-	number (path id)
PathFromLeft	+	-	number (path id)
PathFromAhead	+	-	number (path id)
PathToRight	+	-	number (path id)
PathToLeft	+	-	number (path id)
PathToAhead	+	-	number (path id)
OppositePath	+	-	number (path id)
ToInter	+	-	number (intersection id)
FromInter	+	-	number (intersection id)
ToCNode	+	-	number (connectionnode id)
FromCNode	+	-	number (connectionnode id)
EntranceAllowed	+	+	flag (True/False)
Row	+	+	number (GiveRow, RowOnLeft, RowOnRight, RowOnBoth, EqualPriority, HaveRow)
LastCarNr	+	-	number (participant id)
FirstCarNr	+	-	number (participant id)

Length gives the pathlength in meters, measured along the center of all segments' DLane[0].

NrSegments gives the number of segments on the path.

TrafficLight gives the current state of the trafficlight at the end of the path. When there is no trafficlight, the result is Absent. The trafficlight can be set with the values Red, Yellow, Green, YellowRed, YellowFlash, Blank. Absent has no meaning for setting the trafficlight.

GreenPhase can be set (and read) to change the green-time duration of the next trafficlight group (on the path of the simulatorcar), if there is one.

YellowPhase can be set (and read) to change the yellow-time duration of the next trafficlight group (on the path of the simulatorcar), if there is one.

YellowRedPhase can be set (and read) to change the yellow-red-time duration of the next trafficlight group (on the path of the simulatorcar), if there is one. Normally this value is not set. If it is set by the user, then an extra phase (simultaneous red and yellow) is added between red and green, as in German traffic lights).

PathFromRight gives the pathnumber of the path that comes from right at the next intersection from the perspective of the current path. When there is not path from right, the return value is -1 or Absent. PathFromLeft gives the path from left and PathFromAhead returns the path from ahead after the intersection, when there exists such a path. PathToRight returns the pathnumber of the path to right (outgoing path at the next intersection), from the perspective of the current path. Analogous are the variables PathToLeft and PathToAhead. OppositePath gives the path number of the path in opposite direction relative to the current path. So, if the current path goes from intersection A to B, OppositePath goes from intersection B to A.

ToInter returns the number of the intersection where the path is going to, if there is one. Otherwise –1 is returned (or Absent). In that case ToCNode should return a value other than –1.

FromInter return the number of the intersection the path is coming from. To determine the number of the most recent intersection that was traversed, the following two, functionally identical, statements can be applied :

Path[Part[MainTarget].PathNr].FromInter, or
Path[Path[Part[MainTarget].PathNr].OppositePath].ToInter
Even simpler is :
Part[MainTarget].FromInter.

If FromInter is Absent then FromCNode should return a valid value.

ToCNode returns the number of the connectionnode where the path is going to, if there is one. Otherwise –1 is returned (or Absent). In that case ToInter should return a value other than –1.

FromCNode return the number of the connectionnode the path is coming from. If FromCNode is Absent then FromInter should return a valid value.

EntranceAllowed is a flag that indicates whether the path is one way. If is it allowed to drive into this path, the result is True. EntranceAllowed may be set by the user. If you want to avoid that traffic turns into a certain path then set:

```
Path[..].EntranceAllowed := False;
```

Row returns the right-of-way regulation at the end of the path relative to other paths. The signs can be overruled by setting this variable. Valid values are GiveRow, RowOnLeft, RowOnRight, RowOnBoth, EqualPriority, HaveRow.

FirstCarNr can only be read. It returns the number of the first participant (from the last intersection) on the respective path.

LastCarNr can only be read. It returns the number of the last participant (from the last intersection, closest to the intersection the path is going to) on the respective path.

11.7 Part object (participant)

This is the participant object. A participant has a large number of variables that can be set and read. A participant is a traffic participant.

Table 10. Variables of objecttype Part

Variable	Read	Set	Value
PartNr	+	-	number: Participant id (unique id)
Velocity	+	+	number: current speed in m/s
Acc	+	-	number: current acceleration in m/s^2
PathNr	+	+	number: path id
NextPathNr	+	+	number: path id
PrevPathNr	+	-	number: path id
LastPathNr	+	-	number: path id
SegmentNr	+	-	number: segment id
NextSegment	+	-	number: segment id
ToInter	+	-	number: intersection id
FromInter	+	-	number: intersection id
ToCNode	+	-	number: connectionnode id
FromCNode	+	-	number: connectionnode id
PrefLane	+	+	number: DLane index
Lane	+	+	number: synbolic constant (to set the lane) or lane id (to read the lane)
LaneType	+	-	number: type of lane
LaneIndex	+	-	number: index of lanetype
LeftEdgeLineType	+	-	number: line type (0..4)
RightEdgeLineType	+	-	number: line type (0..4)
OnInterPlane	+	-	flag (True/False)
OnRoundabout	+	-	flag (True/False)
LatPos	+	+	number: either positive (to the left of centerline through DLane[0]) or negative (to the right of centerline through DLane[0])
PrefLatPos	+	+	number: either positive (to the left of centerline through DLane[0]) or negative (to the right of centerline through DLane[0])
WheelBase	+	+	number: meters
CarLength	+	+	number: meters
CarWidth	+	+	number: meters
NextBusStop	+	-	number: bus stop id
DisToBusStop	+	-	number: distance in meters
DisToStopSign	+	-	number: distance in meters
DisToStopLine	+	-	number: distance in meters
DisToVOP	+	-	number: distance in meters
DisToSegment	+	-	number: distance in meters
DisToRealInter	+	-	number: distance in meters
DisToInterCenter	+	-	number: distance in meters
DisToInter	+	+	number: distance in meters
DisFromInter	+	+	number: distance in meters
DisToNextNode	+	-	number: distance in meters
IntersectionTrackLength	+	-	number: distance in meters
RemoveOnDistance	+	+	number: distance in meters
DisFromMain	+	-	number: distance in meters
Route	+	+	number or symbolic constant: <pathid, Left, Right, Straight> when set. Pathid when read
RouteIndex	+	-	number >= 0
RouteLength	+	-	number: distance in meters
RouteLenghtLeft	+	-	number: distance in meters
NextTurn	+	+	number: Left, Right, Straight
RoundaboutDir	+	+	number: Left, Right, Straight
TurnAtEnd	+	+	flag (True/False)
MaxVelocity	+	+	number: m/s
CurrentMaxVelocity	+	-	number: m/s. The current maximum velocity as defined by infrastructue (road, signs) or MaxVelocity
MaxDec	+	+	number: m/s^2

MaxAcc	+	+	number: m/s^2
CarType	+	+	cartype from cartypes.def, zerobased index (f.e. CarType 0 is the first car defined in cartypes.def)
DistanceDriven	+	+	Distance driven in current simulation
AlarmOnMaxVelocity	+	+	True or False. Normally the alarmlights are ON if MaxVelocity has been set to 0, except when AlarmOnMaxVelocity := False
ViewDistance	+	+	Viewing ahead distance for sensor (300 meters default)
StopDis	+	+	number: meters
Rt	+	+	number: time in seconds
Heading	+	+	number: vehicle heading in degrees with repect to the world
MaxG	+	+	number: for example 0.25
TTC	+	-	number: time in seconds
THW	+	-	number: time in seconds
DisToRightEdgeLine	+	-	number: distance in meters
DisToLeftEdgeLine	+	-	number: distance in meters
DisToRightLaneEdge	+	-	number: distance in meters
DisToLeftLaneEdge	+	-	number: distance in meters
PositionOnRoad	+	-	number: 1..3
LeadCar	+	-	number: participant id
RearCar	+	-	number: participant id
ApprCar	+	-	number: participant id
LeftCar	+	-	number: participant id
RightCar	+	-	number: participant id
StraightCar	+	-	number: participant id
DisToLeadCar	+	-	number: distance in meters
DisToRearCar	+	-	number: distance in meters
DisToApprCar	+	-	number: distance in meters
FirstLeadOnMyLane	+	-	number: participant id
FirstLeadOnRightLane	+	-	number: participant id
FirstLeadOnRightLane2	+	-	number: participant id
FirstLeadOnLeftLane	+	-	number: participant id
FirstLeadOnLeftLane2	+	-	number: participant id
FirstRearOnMyLane	+	-	number: participant id
FirstRearOnRightLane	+	-	number: participant id
FirstRearOnRightLane2	+	-	number: participant id
FirstRearOnLeftLane	+	-	number: participant id
FirstRearOnLeftLane2	+	-	number: participant id
FirstApprOnMyLane	+	-	number: participant id
FirstApprOnRightLane	+	-	number: participant id
FirstApprOnLeftLane	+	-	number: participant id
FirstApprOnLeftLane2	+	-	number: participant id
SecondLeadOnRightLane	+	-	number: participant id
DisToFirstLeadOnMyLane	+	-	number: distance in meters
DisToFirstLeadOnRightLane	+	-	number: distance in meters
DisToFirstLeadOnRightLane2	+	-	number: distance in meters
DisToFirstLeadOnLeftLane	+	-	number: distance in meters
DisToFirstLeadOnLeftLane2	+	-	number: distance in meters
DisToFirstRearOnMyLane	+	-	number: distance in meters
DisToFirstRearOnRightLane	+	-	number: distance in meters
DisToFirstRearOnRightLane2	+	-	number: distance in meters
DisToFirstRearOnLeftLane	+	-	number: distance in meters
DisToFirstRearOnLeftLane2	+	-	number: distance in meters
DisToFirstApprOnMyLane	+	-	number: distance in meters
DisToFirstApprOnRightLane	+	-	number: distance in meters
DisToFirstApprOnLeftLane	+	-	number: distance in meters
DisToFirstApprOnLeftLane2	+	-	number: distance in meters
DisToSecondLeadOnRightLane	+	-	number: distance in meters
GuidedSpeedDif	+	-	number: speed in m/s
RequiredSpeedMax	+	-	number: speed in m/s
RuleMaxVelocity	+	+	flag (True/False)
RuleFollow	+	+	flag (True/False)
RuleAdaptToCurve	+	+	flag (True/False)
RuleOvertaken	+	+	flag (True/False)
RuleRowLeft	+	+	flag (True/False)
RuleRowRight	+	+	flag (True/False)

RuleRowStraight	+	+	flag (True/False)
RuleOvertaking	+	+	flag (True/False)
RuleEmergLeft	+	+	flag (True/False)
RuleEmergRight	+	+	flag (True/False)
RuleEmergStraight	+	+	flag (True/False)
RuleRedTrafficLight	+	+	flag (True/False)
RuleYellowTrafficLight	+	+	flag (True/False)
RuleApproachOnMyLane	+	+	flag (True/False)
RuleBusStop	+	+	flag (True/False)
AllowPassRight	+	+	flag(True/False) passing right is allowed (default False)
RuleAdaptToMergingLead	+	+	flag (True/False) let a merging leadvehicle merge in if it uses the indicator (by slowing down), default True
GiveWayToMergingLead	+	+	flag (True/False) move to the next lane if there's a merging lead vehicle that uses the indicator. Default False.
FrontSensor	+	+	flag (On/Off)
RearSensor	+	+	flag (On/Off)
InterSensor	+	+	flag (On/Off)
ApproachSensor	+	+	flag (On/Off)
UseBrakeLight	+	+	flag (On/Off)
BrakeLight	+	+	flag (On/Off)
UseIndicator	+	+	flag (On/Off)
Indicator	+	+	symbolic constants: IndicatorOff, IndicatorLeft, IndicatorRight, IndicatorAlarm
SwingPhase	+	+	number: time in seconds
SwingAmplitude	+	+	number: distance in meters
Xpos	+	-	number: coordinate position
Ypos	+	-	number: coordinate position
IsPriorityVehicle	+	+	number: (True/False). sets or reads whether it is a priority vehicle (use of siren). A priority vehicle is treated differently by other traffic.
InList	+	-	flag (True/False)
IsdType	+	+	number: type
IsdCat	+	-	number
RoadOrder	+	+	The type of road the participant is on: 1=CountryRoad, 2=MotorRoad, 3=Highway, 4=UrbanArea
DumVar0	+	+	number
DumVar1	+	+	number
DumVar2	+	+	number
DumVar3	+	+	number
DumVar4	+	+	number
DumVar5	+	+	number
DumVar6	+	+	number

PartNr is the participant number of the car. This can be used to access each participant individually. The simulator car can be accessed as MainTarget of 0, f.i. Part[MainTarget].'Variable', or Part[0].'Variable'. The id cannot be set: it is created by the system in PNr := CreatePartIsd(); In that case PNr is the unique PartNr.

Velocity is the current speed in m/s. It can be set to give a participant an initial speed or read.

Acc is the current acceleration in m/s². Can only be read.

PathNr is the current pathnumber. This is an important variable that can also be set. To reposition a participant (f.i. to give it an initial position), a PartNr must be set, in combination with either DisFromInt or DisToInt. There are no other ways of longitudinal positioning.

NextPathNr is the number of the next path.

PrevPathNr is the number of the previous path.
LastPathNr is the same as PrevPathNr.

SegmentNr is the number of the current segment.

NextSegment is the number of the next segment on the current path. If there is ony one segment on the path, or if the participant is on the last segment of the current path, the result is -1 or Absent.

ToInter returns the number of the next intersection, or –1 (Absent) if there is none.

FromInter returns the number of the last past intersection, or –1 (Absent) if there was none.

ToCNode returns the number of the next connectionnode, or –1 (Absent) if there is none.

FromCNode returns the number of the last past connectionnode, or –1 (Absent) if there was none.

PrefLane refers to the preferred lane in which the participant drives. It can be set as the preferred DLane index: f.i. 0 = DLane[0] (the rightmost normale driving lane, 1 = DLane[1] (the lane to the left of DLane[0]) etc. If it is read te preferred DLane index is returned.

Lane gives the lane id on which the participant is (center of front bumper). Lane can also be used to change the lateral position of the participant. In that case there occurs a repositioning to the center of the respective lane. To set the Lane the symbolic constants RightLane, LeftLane of RightShouder are used. So, f.i. Part[..].Lane := RightLane and LaneId := Part[..].Lane. Alternatively, a lane id can be used to set the Lane. This is only executed if the lane id is a lane on the current segment.

LaneType returns the current LaneType: DLane, ExitLaneRight, ExitLaneLeft, EntryLaneRight, EntryLaneLeft, HardShoulder, HardShoulderLeft, BicycleLaneRight, BicycleLaneLeft, PavementRight or PavementLeft.

LaneIndex returns the current lane index. If there are 2 DLanes on the present segment, and the participant is driving in the left lane then LaneIndex is 1.

LeftEdgeLineType returns the line type of the right edgelane of he current lane. 0 = none, 1 = Continuous, 2 = one-three, 3 = Blockmarkings, 4 = three-nine

RightEdgeLineType returns returns the line type of the left edgelane of he current lane 0 = none, 1 = Continuous, 2 = one-three, 3 = Blockmarkings, 4 = three-nine

OnInterPlane returns a flag (True/False) indicating whether the participant is on the intersection plane.

OnRoundabout returns a flag (True/False) indicating whether the participant is on a roundabout complex.

LatPos gives the lateral position in meters. It can also be used to change the lateral position. In that case tge new lane is automatically computed and the particiant is moved laterally. A negative lateral position means that the car is to the right of the centerline of DLane[0]. A positive lateral position means that the car is to the left of the centerline of DLane[0]. A lateral position of 0 indicates that the center of the car is precisely in the middle of the rightmost driving lane (DLane[0]).

PrefLatPos gives the preferred lateral position. It can also be used to set a preferred lateral position. Normally this is 0. A positive value lets the participant drive more to the left of the centerline of DLane[0], while a negative lateral position results in driving to the right of this centerline.

WheelBase sets or reads the wheelbase of the participant. This is the distance between the front and rear wheel axes (in meters).

CarLength sets or reads the total length of the participant (in meters).

CarWidth sets or reads the width of the participant (in meters).

NextBusStop returns a bus stop id if a bus stop is approached. It returns Absent if these is none.

DisToBusStop returns the distance (along the path) to the next bus stop. It returns 9999 if a busstop could not be found

DisToStopSign returns the distance (along the path) to the next stopsign on the route. It returns 9999 if a stopsign could not be found

DisToVOP returns the distance (along the path) to the zebra crossing on the route. It returns 9999 if a zebra crossing could not be found

DisToSegment returns the distance to the start of the next segment in meters.

DisToRealInter returns the distance to the next 'real' intersection (along the route). A real intersection is an intersection that is not a 'virtual' intersection (with only two branches) and not a connectionnode.

DisToInterCenter gives the distace along the path to the center point of the next intersection, while all virtual intersection in-between are skipped.

DisToInter gives the distance to the start of the next node in meters. This is the distance to the end of the current path. If a 'real' intersection is at the end of the current path then the distance to the beginning of the intersectionplane is computed. Setting this variable must occur in combination with setting PathNr. The order in which the assignments occur is not significant.

DisFromInter gives the distance along the path from the last intersection. Setting this variable must occur in combination with setting PathNr. The order in which the assignments occur is not significant.

DisToNextNode gives the distance along the path to the next routenode.

IntersectionTrackLength gives the length of the total track on the next intersection, depending on the route, if there is a next intersection one (else IntersectionTrackLength = 0).

RemoveOnDistance gives and sets the absolute distance (in meters) from the simulator car (MainTarget) at which the participant is removed and deleted. This mechanism ensures that the participant is removed when it gets too far away to be seen.

DisFromMain gives the absolute distance (in meters) from the simulator car (MainTarget).

Route sets the route of a participant. It can be assigned the values <Left, Right, Staight> or a path id, and the values Clear and StoreRoute. Clear means that the route is cleared. StoreRoute means that the specification of the route is ready and the route pointer is initialized. Route appends the values to a route. For example :

```
// first position participant
Part[].PathNr := 21;
Part[].DisToInter := 50;
// then build route
Part[].Route := Clear;
```

```
Part[].Route := 12;
Part[].Route := 14;
Part[].Route := 8;
Part[].Route := StoreRoute;
```

All paths in the route must connect to each other. Also, the first path in the route (12 in the example) should connect to the current path (21 in the example).
If Route is read then the next pathid on the route is returned.

RouteIndex returns the current (zero based) routeindex).

RouteLength returns the total length (in meters) of the route.

RouteLengthLeft returns the length (in meters) of the route from the present position

NextTurn can be set or read with the values Right, Left or Straight.

RoundaboutDir can be set or read with the values Right, Left or Straight. Right = take first exit, Straight = take second exit, Left = three quarters.

TurnAtEnd can be set or read with a flag (True/False). It is is True then the participant turns around at the end of a deadend street (intersection with only 1 branch). If False, it stops at the end of a deadend street.
MaxVelocity returns and sets the maximum allowed speed in m/s. This variable can be set in order to control the speed of the car. This is the velocity the car strives for if it is not limited by other rules.
MaxDec returns and sets the maximum allowed deceleration in m/s^2. When this is higher, the car brakes harder and starts braking at a shorter distance from an object.

MaxAcc returns and sets the maximum allowed acceleration in m/s^2.

StopDis returns and sets the distance (in meters) the participants adds to the distance to an object if stops. If, for example, StopDis is set to 1 meter, then the participant stops 1 meter before a stopline or before the beginning of an intersection plane.

Rt is the reaction time of the virtual driver of a participant. This value is used in a number of behavioural rules

MaxG is the amount of G accepted by the virtual driver of the oparticipant in driving curves. For example, a value of 0.25 means that the maximum G-force accepted is 0.25*. With higher values, velocity in curves is higher. In addition to this, this factor is moduled as a function of the curve radius.

TTC is the time-to-collision (in seconds) to the first lead participant in the same lane as the participant.

THW is the timeheadway (in seconds) to the first lead participant in the same lane as the participant.

DisToRightEdgeLine is the distance (in meters) between the right side of the participant and the right side of the road.

DisToLeftEdgeLine is the distance (in meters) between the left side of the participant and the left side of the road.

DisToRightLaneEdge is the distance (in meters) between the right side of the participant and the right side of the present lane.

DisToLeftLaneEdge is the distance (in meters) between the left side of the participant and the left side of the present lane.

PositionOnRoad gets the present roadposition: 1= OnRoad, 2 = OffRoadRight, 3 = OffRoadLeft.

LeadCar is the participant id of the first leadvehicle (not neccessarily in the same lane). Absent if none.
RearCar is the participant id of the first rearvehicle (not neccessarily in the same lane). Absent if none.
ApprCar is the participant id of the first approaching vehicle on the same road (not neccessarily in the same lane). Absent if none.
LeftCar is the participant id of the first approaching vehicle from left at the next intersection. Absent if none.
RightCar is the participant id of the first approaching vehicle from right at the next intersection. Absent if none.
StraightCar is the participant id of the first approaching vehicle from ahead at the next intersection. Absent if none.

DisToLeadCar gives the bumper to bumper distance to the first leadvehicle, measured along the path of the car.
DisToRearCar gives the bumper to bumper distance to the first rearvehicle, measured along the path of the car.
DisToApprCar gives the bumper to bumper distance to the first approaching vehicle, measured along the path of the car.

FirstLeadOnMyLane is the participant id of the first leadvehicle in the same lane as the participant. Absent if none.
FirstLeadOnRightLane is the participant id of the first leadvehicle in the first lane right of the lane the participant is in. Absent if none.
FirstLeadOnRightLane2 is the participant id of the first leadvehicle in the second lane right of the lane the participant is in. Absent if none.
FirstLeadOnLeftLane is the participant id of the first leadvehicle in the first lane left of the lane the participant is in. Absent if none.
FirstLeadOnLeftLane2 is the participant id of the first leadvehicle in the second lane left of the lane the participant is in. Absent if none.
FirstRearOnMyLane is the participant id of the first rearvehicle in the same lane as the participant. Absent if none.
FirstRearOnRightLane is the participant id of the first rearvehicle in the first lane right of the lane the participant is in. Absent if none.
FirstRearOnRightLane2 is the participant id of the first rearvehicle in the second lane right of the lane the participant is in. Absent if none.
FirstRearOnLeftLane is the participant id of the first rearvehicle in the first lane left of the lane the participant is in. Absent if none.
FirstRearOnLeftLane2 is the participant id of the first rearvehicle in the second lane left of the lane the participant is in. Absent if none.
FirstApprOnMyLane is the participant id of the first approaching vehicle in the same lane as the participant. Absent if none.
FirstApprOnRightLane is the participant id of the first approaching vehicle in the first lane right of the lane the participant is in. Absent if none.
FirstApprOnLeftLane is the participant id of the first approaching vehicle in the first lane left of the lane the participant is in. Absent if none.
FirstApprOnLeftLane2 is the participant id of the first approaching vehicle in the second lane left of the lane the participant is in. Absent if none.
SecondLeadOnRightLane is the participant id of the second leadvehicle in the first lane right of the lane the participant is in. Absent if none.

DisToFirstLeadOnMyLane gives the bumper to bumper distance to the first leadvehicle in the same lane as the participant. Only valid if FirstLeadOnMyLane != Absent.
DisToFirstLeadOnRightLane gives the bumper to bumper distance to the first leadvehicle in the first lane right of the lane the participant is in. Only valid if FirstLeadOnRightLane != Absent.
DisToFirstLeadOnRightLane2 gives the bumper to bumper distance to the first leadvehicle in the second lane right of the lane the participant is in. Only valid if FirstLeadOnRightLane2 != Absent.
DisToFirstLeadOnLeftLane gives the bumper to bumper distance to the first leadvehicle in the first lane left of the lane the participant is in. Only valid if FirstLeadOnLeftLane != Absent.
DisToFirstLeadOnLeftLane2 gives the bumper to bumper distance to the first leadvehicle in the second lane left of the lane the participant is in. Only valid if FirstLeadOnLeftLane2 != Absent.
DisToFirstRearOnMyLane gives the bumper to bumper distance to the first rearvehicle in the same lane as the participant. Only valid if FirstRearOnMyLane != Absent.
DisToFirstRearOnRightLane gives the bumper to bumper distance to the first rearvehicle in the first lane right of the lane the participant is in. Only valid if FirstRearOnRightLane != Absent.
DisToFirstRearOnRightLane2 gives the bumper to bumper distance to the first rearvehicle in the second lane right of the lane the participant is in. Only valid if FirstRearOnRightLane2 != Absent.
DisToFirstRearOnLeftLane gives the bumper to bumper distance to the first rearvehicle in the first lane left of the lane the participant is in. Only valid if FirstRearOnLeftLane != Absent.
DisToFirstRearOnLeftLane2 gives the bumper to bumper distance to the first rearvehicle in the second lane left of the lane the participant is in. Only valid if FirstRearOnLeftLane2 != Absent.
DisToFirstApprOnMyLane gives the bumper to bumper distance to the first approaching vehicle in the same lane as the participant. Only valid if FirstApprOnMyLane != Absent.
DisToFirstApprOnRightLane gives the bumper to bumper distance to the first approaching vehicle in the first lane right of the lane the participant is in. Only valid if FirstApprOnRightLane != Absent.
DisToFirstApprOnLeftLane gives the bumper to bumper distance to the first approaching vehicle in the first lane left of the lane the participant is in. Only valid if FirstApprOnLeftLane != Absent.
DisToFirstApprOnLeftLane2 gives the bumper to bumper distance to the first approaching vehicle in the second lane left of the lane the participant is in. Only valid if FirstApprOnLeftLane2 != Absent.
DisToSecondLeadOnRightLane gives the bumper to bumper distance to the second leadvehicle in the first lane right of the lane the participant is in. Only valid if DisToSecondLeadOnRightLane != Absent.

GuidedSpeedDif gives the speeddifference between the curren speed and the maximum allowed speed according to the normative rules set. If this value is positive than the simulator driver is driving too fast in relation to a set of normative rules. Speed difference is in m/s.

RequiredSpeedMax is the maximum speed allowed according to a set of normative rules. In m/s.

RuleMaxVelocity can be switched on or off. If it is off, then the maximum velocity of the autonomous agent (participant) is no longer controlled by traffic signs or area (buildup area, maximum speed signs, highway signs and so on). The maximum velocity in that case is only controlled by the preferred maximum velocity of that car (MaxVelocity) or by curves in the road and other traffic.
RuleFollow switches the rules for car following on or off.
RuleAdaptToCurve switches the rules for speed control in curves on or off.
RuleOvertaken switches the rules for being overtaken on or off.
RuleRowLeft switches the rules for speedcontrol for traffic from left on or off.

RuleRowRight switches the rules for speedcontrol for traffic from right on or off.
RuleRowStraight switches the rules for speedcontrol for traffic from ahead after intersections on or off.
RuleOvertaking swithces the rules for overtaking on or off.
RuleEmergLeft switches the emergency procedures for speedcontrol to traffic from left on or off.
RuleEmergRight switches the emergency procedures for speedcontrol to traffic from right on or off.
RuleEmergStraight switches the emergency procedures for speedcontrol to traffic from ahead after intersections on or off.
RuleRedTrafficLight switches the rules for red traffic lights on or off.
RuleYellowTrafficLight switches the rules for yellow traffic lights on or off.
RuleApproachOnMyLane switches the rules for speedcontrol to oncoming traffic on or off.
RuleBusStop switches the rule for autonomous stopping for bus stops on or off (default off: should be switched on for buses.
FrontSensor switches the perception of leadvehicles on or off.
RearSensor switches the perception of rearvehicles on or off.
InterSensor switches the perception of traffic approaching an intersection on or off.
ApproachSensor switches the perception of oncoming vehicles on or off.

UseBrakeLight is a flag to set the use of the brakelights on of off. If it is Off, then the participants does not light up the brakelight when it brakes. If it is On, then the participant uses the brakelights if necessary.
BrakeLight is a flag to switch the brakelights on, independenly from the brakelight control of the autonomous agent.
UseIndicator is a flag to set the use of the direction indicators on of off. If it is Off, then the participants does not apply its direction indicators. If it is On, then the participant applies direction indicators if necessary.
Indicator may be read. If set it is used to apply the direction indicators, independently from the indicator control of the autonomous agent. Possible values are IndicatorOff, IndicatorLeft, IndicatorRight or IndicatorAlarm.

SwingPhase can be set and read. It represents the time the car takes to swerve from a certain lateral position to another one. Normally this is a random process. It can be set however to fully control the time it takes to swerve to another lateral position. If set to zero, the car does not swerve anymore.

SwingAmplitude can be set and read. Represents the lateral distance (from the preferred lateral position) within which the car swerves as a random process. If set to zero, the car does not swerve anymore.

Xpos gives the X coordinate position of the car

Ypos gives the Y coordinate position of the car

InList can only be read and returns True or False depending on whether the car is in the current traffic list.

IsdType is a type of vehicle wit a specific brand and color

IsdCat is a category of vehicle (f.i. a bus = 20, a bicycle = 30 and a pedestrian = 40).

DumVar0 to DumVar4 are variables that can be read and set to store a temporary value. These variables serve as buffers to store information

12 Suggestions for debugging

- When debugging the script files, the following steps are important:

1) locate the line in scentemp##001 of the first error.
2) find this error in your script source files
3) fix the error
4) run the script again to test for errors

Do this error by error.

- A frequent type of error is the use of a reserved keyword as a variablename, as in

```
Var { SwingPhase; }
```

This is illegal because SwingPhase is a reserved keyword.

- It is important to adhere to the following order of blocks:

```
Define Scen[..] {
        Var { ..;..; }
        Start {
                …
        }
        Do {
                …
        }
        End {
                …
        }
        // followed by a list of actions
        Define Action[0] {
                Start {
                        …
                }
                End {
                        …
                }
        }
        Define Action[1] {
                Start {
                        …
                }
                End {
                        …
                }
        }
}
```

- Keep in mind that userdefined variables have either local scope or global scope. If a variable has global scope, it is defined outside a scenario. If it has local scope it can be defined anywhere within a scenario, including an action. So local scope refers to ‘local within the scenario’ and not to ‘local within an action’.

```
Define Scen[..]  {
        Define Action[0] {
                Var { ThisVar; }
        }
        Define Action[1] {
                Var { ThisVar; } // illegal because ThisVar has already been defined
```

```
        }
}
```

- Make use that all scenarios have a unique ID. Also make sure hat all actions defined within a scenario have a unique ID. So the following may result in problems:

```
Define Scen[100] {

}

Define Scen[101] {

}

Define Scen[100] {  // Scen[100] has already been defined

}
```