Preface

This document describes the basics of the cognitive architecture CASCaS. For which purpose was it developed? What are typical scientific or application questions where CASCaS can be used to model, simulate and analyze problems? What are the main theories and modeling concepts behind the architecture? What needs to be done to use the architecture?

The architecture was originally developed by Andreas Lüdtke as a substantial part of his PhD thesis (Lüdtke, 2002). At that time it was written in the programming language Prolog. During the last 7 years CASCaS was completely re-coded by different members of the OFFIS group Human-Centered Design and now it is a plain C application which is compiled using GNU C compiler and it also makes extensive use of the GNU programming libraries GTK+ and glib. For some visualizations Graphviz is used as external tool (and must be installed separately, because it is not part of CASCaS). We currently work on a replacement for this library based on OpenGL.
# Table of Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 The main components of CASCaS</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Basics</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Data Structures</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 Procedural Knowledge</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Perception</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Visual Perception</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Auditory Perception</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Haptic Perception</td>
<td>9</td>
</tr>
<tr>
<td>2.2.4 3D EGO-centric coordinate system</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Goal Module &amp; Associative Layer</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Goal Selection &amp; Execution</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Rule Selection &amp; Execution</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Autonomous Sensory and Motor Patterns</td>
<td>14</td>
</tr>
<tr>
<td>2.5 CASCaS Procedure Language</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1 Specification of declarative knowledge</td>
<td>16</td>
</tr>
<tr>
<td>2.5.2 Rule specification</td>
<td>18</td>
</tr>
<tr>
<td>2.5.3 Basic Module Initialisation Statements</td>
<td>24</td>
</tr>
<tr>
<td>2.5.4 Preprocessor Statements</td>
<td>24</td>
</tr>
<tr>
<td>2.5.5 Example Procedure File</td>
<td>25</td>
</tr>
<tr>
<td>3 CASCaS Simulation Tutorial</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Data Definition File</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Configuration file</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Simulation Environment Event Scheduler</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1 Script File</td>
<td>36</td>
</tr>
<tr>
<td>4 Bibliography</td>
<td>36</td>
</tr>
</tbody>
</table>
1 Introduction

The Cognitive Architecture for Safety Critical Task Simulation (CASCaS) is a framework for modeling and simulation of human behavior. It is an executable program\(^1\) and simulates human behavior. CASCaS uses an internal simulation clock and can be used in real-time or accelerated-time simulation. It can be connected to different simulation platforms, e.g. flight simulator or driving simulator environments using either UDP socket connections or IEEE HLA-2000 simulation standard. For the latter one we currently use the open source RTI CERTI\(^2\), if you are interested in this please contact us for further information. Besides the interfaces to connect with different simulation platforms CASCaS comes with a simple step-by-step event scheduler, which is useful for creation, testing and debugging of initial model versions. More about the possibilities to connect to different environments can be found in section 3.

Because CASCaS is a cognitive architecture, it is based on a number of psychological theories about human cognition and also incorporates physiological models and limits. Its purpose is to model and simulate human machine interaction in safety-critical domains like aerospace or automotive but in general it is not limited to those specific domains. The first application was a pilot model which was built by Lüdtke (Lüdtke, 2002) to simulate learned carelessness errors. The cause for this type of error is a simplification of procedural actions that should be executed, e.g. specific monitoring actions and checks to observe the state of an autopilot system. Typically, systems like the autopilot or other cockpit systems work as intended, they very rarely produce any errors. Monitoring actions become routine and might lead to deviations of the behavior: some checks are likely to be neglected, if the expected outcome always is the same and the system never seems to fail. Learned carelessness is one theory that tries to explain these effects and was analyzed and modeled in CASCaS.

Over the last years the Human-Centered Design group at OFFIS developed several different models for the aeronautics domain as well as the automotive. Details of those models can be found in various publications (Lüdtke & Osterloh, 2009), (Frische & Osterloh & Lüdtke, 2010), (Wortelen & Baumann & Lüdtke, in press), (Wortelen & Lüdtke & Baumann, 2013), (Weber & Steenken & Lüdtke, 2013). The development of those models lead to extensions of the architecture itself, hence they are part of the current state of CASCaS.

This document will provide the basic documentation for all components of CASCaS, how they process data and how the overall information processing is done. It will not go into all the details about each component and the underlying psychological theories. In addition to the basic information how CASCaS works, the configuration and input files which are necessary to start a simulation are described. Finally, the generated output files and traces are explained which can be used for analysis of the behavior that the model showed during a simulation run.

\(^1\) Currently only a windows version exists, linux or mac versions are not available
\(^2\) https://savannah.nongnu.org/projects/certi
2 The main components of CASCaS

Figure 1 shows the current version of the architecture with all its components. This chapter will explain their functionality and the interaction between the components that leads to the overall behavior which can be observed during the simulation. Basically, the architecture consists of 5 components: a Goal Module which stores the intentions of the model what it wants to do next. The Central Processing is subdivided into three different layers: the cognitive layer which can be used to model problem solving, the associative layer executes learned action plans and the autonomous layer simulates highly learned behavior. The memory component is subdivided into a procedural (action plans) and a declarative knowledge (facts) part. The Perception component contains models about physiological characteristics of the visual, acoustic and haptic sensory organs, for example models about peripheral and fovea vision. To interact with the external environment the Motor component of CASCaS contains models for arm, hand and finger movements. It also comprises a calculation for combined eye / head movements that are needed to move the visual perception to a specific location. In general the model starts observing its environment with the perception and receives input that is stored in the declarative memory. Dependent on its current intention and the perceived environment it selects action plans and tries to achieve its current goal. It may generate new goals and further actions, which can be triggered by events perceived from the environment or it the model may itself create new goals based on its own decision making process to initiate a certain behavior.

Before we can start with the introduction of the different modules it is necessary to understand the basic data structures that are processed within CASCaS (section 2.1). Afterwards, the different perception channels (2.1.2) are explained which perceive objects and store them in the memory component for further processing steps. The most important part of the behavior that CASCaS can reproduce is based on the goal module and the associative layer, which are explained in subsection 2.3. The motor component which CASCaS uses to manipulate control elements of the environment is presented in subsection 2.4. In the sections afterwards a detailed explanation of the procedure language elements is given including some more rule examples and also the necessary files to set up a simulation are described.

2.1 Basics

2.1.1 Data Structures

This short section does not explain the complete syntax of the data specification file (file ending: .xtop), it just gives the information which are necessary to understand how CASCaS deals with data respectively objects. The full syntax of the data structures is given in section 3.1. If you want to model and simulate with CASCaS you need to set up some basic input files. One of them specifies the data
types that can be used in the simulation. The following short xml code fragment shows an abstract data type definition which is necessary to define the structure of an object that CASCaS can use:

```xml
<tns:type name="Car" modality="visual">
  <tns:variable name="id" type="int" />
  <tns:variable name="x" type="double" />
  <tns:variable name="y" type="double" />
  <tns:variable name="z" type="double" />
  <tns:variable name="speed" type="double" />
  <tns:variable name="distance" type="int" />
</tns:type>
```

In the example above you can see that an abstract type Car is created. The modality attribute defines by which perception channel this object can be perceived. In this case the car is perceivable by the visual perception. The data type has some attributes (\(<\text{tns:variable ... } />\)): the first four \(\{id, x, y, z\}\) are necessarily needed in every object definition. The 3D coordinate is primarily used for calculation of gaze directions and eye / head movement, the use of the id is explained later on. Besides these attributes any number of additional attributes can be added, in this case two attributes speed and distance are specified. Each attribute also has a datatype which can be either int, float, double or string. All data types that you define can be understood as schemata for concrete object instances that CASCaS receives at runtime. Typically, CASCaS is connected to some kind of simulator, e.g. a driving or a flight simulator. For this purpose you define some typed input and output channels which are used to receive and send data. This can be seen in the next short code snippet:

```xml
<variable_definition>
  <variable name="car_1" type="Car" mode="in">
    <options> </options>
  </variable>
  <variable name="keys" type="Keyboard" mode="out">
    <options> </options>
  </variable>
</variable_definition>
```

The \(<\text{variable_definition}>\) section contains specifications for input and output channels through which CASCaS can receive and send typed data. The channel definition is also used by the so called datapool of CASCaS that manages input and output objects: with this definition CASCaS now has one input channel for objects of the type "Car". This channel has the attribute mode="in", which means that data is send to CASCaS from extern via this channel and CASCaS can perceive it. The external simulator has to assure that he sends the appropriate data according to the data type specification.

In the data type section we also defined modality="visual" for the type "Car": therefore, if an object is received through this input channel, the datapool of CASCaS notifies the visual perception component. If the object is located within the visual field it can be perceived and is written into the memory component. The second channel is an output channel (mode="out"). This indicates that CASCaS can send output to a simulator via his motor component. The data type definition for Keyboard is not specified in this example but one can imagine an object with attributes like key1, key2 and so on which could be pressed by the model. In case of a driver model one could define output objects for the steering wheel or the accelerator and brake pedals.

A final, but very important remark has to be made about the name attribute: the name of the channel does not have any "cognitive plausible" meaning to CASCaS, respectively the internal cognitive components. This name is used only for the specification of channels to communicate with an external simulator. Example: if CASCaS receives a car object via the input channel defined above it may visually perceive the object if it is located inside the visual field. But this does not mean that CASCaS has any deeper insight about the object and can associate automatically what for example a "leadcar" is. It is a pitfall to assume that you could name the channel "leadcar" or "car behind" and assume that the model knows what a leadcar is. The channel name is not part of the cognitive modeling and it will not appear in the models memory component. The model needs additional declarative knowledge.

---

3 Currently we do not allow different perception channels for different attributes of one data type. This maybe changed in a future revision of CASCaS. So far if you want to specify an acoustic input you have to define a second data type, which could be named motorsound for example.
(explained later in section 2.5.1) that it has learned about objects in his environment. To know what a leadcar is it needs to know what his own lane is and what that the "closest" car "ahead" on his own lane is. With all this knowledge the symbolic name "leadcar" can be associated with an object. The model needs to match actual perceptions with his learned knowledge. This is what is often referred to as modeling situation awareness level 1 and 2 (Endsley 1995, 1995a).

Besides the knowledge about data structures that CASCaS uses, you also need to know a little about procedural knowledge, which is explained in the following very short section.

2.1.2 Procedural Knowledge

In the introductory part the associative layer was already mentioned. This layer executes action plans that are stored in a rule-based format. We do not go into the details at this point, it is sufficient to know that the associative layer selects and executes amongst a number of rules that are specified in the procedure file (file ending .proc). In the example code fragment the basic parts of a rule specification are shown. Each rule has a conditional part that defines a search pattern across the content of the memory component of the model. The conditional part has to be fulfilled successfully, before the action part can be executed. The action part consists of a number of statements below the "-->

```plaintext
//attributes and items of a rule
rule(head){
  Condition(boolean expression)
  -->
  LookAt(object)
  Motor(resource, type, instrument, value)
}
```

The LookAt command requires an object as parameter which the model can receive through an input channel. Also the modality of the data type must be "visual" as it was defined in the data type example above. The Motor statement is used to manipulate objects in the environment. It contains some parameter: the resource can be either left hand or right hand. The type specifies different kinds of movements like grasping or adjusting. The detailed type commands are explained in section 2.4. The instrument is the object that should be manipulated and value defines the new value that should be typed / dialed in. The instrument / object has to be defined as a data type which is associated with an output channel. As an example the object could be a Keyboard with a number of keys that the model can press. The Keyboard has to be defined as datatype and an output channel also has to be specified.

With these brief explanations about data types, objects and the main processing entity of the associative layer (the rules) we can now start to explain the different modules of CASCaS and the data flow between them. We start with the input modules (perception) and finish with the output (motor) of the model.

2.2 Perception

CASCaS contains implementations for different input modalities. Since the first version of CASCaS a visual perception (see 2.2.1) was integrated which was successively improved and contains models for fovea and peripheral vision. The auditory channel (see 2.2.2) can receive and subdivide between speech and tone messages. A basic approach for multimodal perception was implemented using the TWIN model concepts (Colonius & Diederich 2004) but this is still basic research and not part of this introductory documentation.

2.2.1 Visual Perception

The visual perception of CASCaS is divided into two components: the fovea vision which implements a small center area of the visual gaze with the sharpest vision. This area is about 2 degree around the center point of the gaze. It is used to perceive detailed information about an object, e.g. reading a text can only be done with the fovea vision. The peripheral visual field is much larger ~170 degree horizontally and 110 degree vertically (Osterloh & Lüdtke 2008). The peripheral vision is used for the detection of movement and is absolutely necessary for reactive behavior. Remember the example
from the data type section with the "Car" object (section 2.1.1). Assume, the datapool of CASCaS perceives a car object from an external driving simulator through the visual input channel. The vision module is triggered by the datapool and starts to calculate if the coordinates of the object lay within the peripheral field of view. If this is the case the object coordinates are stored in the memory component, which is organised as a semantic network (see Figure 2).

Figure 2: object stored into the memory by the visual perception. The node in the middle (destination of the "car" association) is called the base node of the object and is derived from the defined type.

Besides the coordinates (x,y,z) there are some other attributes which are explained next: isa, id and attended. Furthermore, there is a link named "car#id=7". This link and also the id attribute are used for internal processing purposes and should under no circumstance be considered for any further modeling issues, especially the link. Why does CASCaS need those two two? So far, we have not implemented any pattern matching algorithms to model human capabilities for object matching and tracking, which are very sophisticated skills of the human cognition. CASCaS currently uses the id as an symbolic abstraction: each object must have a unique id, which is used to "track" changes for a specific object. The id is an artificial construct which is only used for this purpose: object matching / tracking. It allows the visual perception to write coordinate updates each simulation step. Because the id is also a very useful debugging attribute to identify objects and their perceived states at runtime, we do not mask it from the visualisation of the memory component. Each new object with a different id is stored in a separate object tree in the memory. The correct numbering (uniqueness) of the ids has to be done in the external simulator environment which provides (sends) those objects to CASCaS.

The example in the figure also shows two default attributes that are stored by the perception component each time a new object is added:

- **attended** (String 'yes' 'no'): is written once when the object is detected the first time. The initial value is set to 'no'. A similar concept exists in the cognitive architecture ACT-R (Anderson et al., 2004)
- **isa** (String): contains the object type as a string representation.

The "attended" attribute is very useful to check for new objects in the visual field. Imagine the model executes a number of tasks and a new car object appears in the peripheral visual field: to force a reaction an additional rule could be implemented, which does not belong to a specific goal and can fire at anytime inbetween the execution of a number of tasks:

```
rule{id=123, type=reactive} {
    Condition(car.attended == 'no')
    -->
```
In this case, the rule will always fire if a car is found in the memory that was not attended previously. The action part of the rule triggers the visual perception to look at the object. It may also trigger further goals and actions which are necessary to deal those objects. In this case the attended flag of the object is switched to "yes" which prevents that the rule will fire again for this object.\footnote{In some situations it may be useful to reset the flag later on. In contrast, ACT-R uses an automatic reset of the attended flag, which is currently not done by CASCaS.}

Each `LookAt` command within a rule shifts the visual focus to the object given in that statement. If the model has shifted its gaze and focusses the object it will read the information available in the fovea vision.

In the example of the data type section (see 2.1.1) the car object has two additional attributes "speed" and "distance". The fovea needs some time to fixate and read those details of an object, at least 200ms. After this period of time additional links are attached to the object with the values of those attributes. At this point we do not care about the cognitive / physiological plausibility of additionally defined attributes, a distance in cm is certainly not what people can estimate. The driver model developed in the project IMoST tackles some of those issues (Weber & Steenken & Lüdtke, 2013).

What happens if the object moves out of the focus which is very likely in a real time driving simulation? To prevent this, the visual perception of CASCaS is able to automatically initiate small (mikro-) saccades for object tracking. Once the object is focussed, a change in the coordinates will force the visual perception to initiate such a saccade to follow the object in 3D space. The peripheral visual field is (obviously) automatically moved as well and is able to detect to detect objects that appear, disappear or move through the visual field. This is a necessary feature for all models that should be able to interact in real world simulations with moving objects.

**Tip:** Some more very important information about the behavior of the visual perception at simulation time are given in tutorial section 3

### 2.2.2 Auditory Perception

In contrast to the visual perception the auditory perception does not contain any psycho-physiological models so far. It should not be used to model specific auditory perception problems like hearing problems, the cocktail party effect or similar phenomena. The auditory channel can be used to recognize auditory signal which are above a certain (adjustable) decibel threshold. A possible application is to implement models which simulate laboratory reaction or choice reaction task experiments. In such experiments, participants have to react to simple acoustic stimuli with a button press for example. Modeling real world tasks, where humans have to deal with acoustic warning signals is also a meaningful application until the goal is not to have a psycho-physiological model if a certain signal is detectable in a certain environment. The following explanations will introduce how the auditory perception can be integrated into a model.

As well as for the visual perception the auditory perception processes objects which are defined in the data type definition file. Let’s assume you want to simulate a multispeaker audio equipment inside a car that is used to generate warning signals for a driver assistance system from different spatial positions. If a CASCaS model should react to those acoustic cues, a data type has to be defined for such input signals:

```xml
<tns:type name="Audiosignal" modality="auditory_tone">
  <tns:variable name="id" type="int" />
  <tns:variable name="x" type="double" />
  <tns:variable name="y" type="double" />
  <tns:variable name="z" type="double" />
  <tns:variable name="frequency" type="double" />
  <tns:variable name="amplitude" type="double" />
</tns:type>
```
Part 1: An Introduction to CASCaS

The statement `modality="auditory_tone"` signalises CASCaS that the object can be perceived by the auditory perception component. Besides the already known attributes id,x,y,z an object which should be perceivable by the auditory perception needs the two attributes `frequency` and `amplitude`. In a simulation you may have to handle auditory signals from different sources at the same time. In that case you need more than one input channel, because each channel can only receive or send one object at the time.

```xml
<variable_definition>
    <variable name="speaker_1" type="Audiosignal" mode="in">
        <options> </options>
    </variable>
    <variable name="speaker_2" type="Audiosignal" mode="in">
        <options> </options>
    </variable>
</variable_definition>
```

The definition above creates two different input channels for objects of the type `Audiosignal`. At this point it is important to remember that the `name` attribute (here `speaker_1`, `speaker_2`) does not mean anything to CASCaS. As it was already said in the section about the data types, the cognitive architecture needs additional declarative knowledge to identify a certain signal (object) as an audio signal coming from a specific source, for example the left stereo speaker. Section 2.5.1, will cover learned declarative knowledge.

As soon as the auditory component receives an `Audiosignal` object, it will add a new object to the memory component similar to the visual perception and attach links for all specified attributes and their values. Also as long as the channels transmits the signal the attributes are updated. In contrast to visual signals, auditory signals can not have any other attributes than the ones specified above and they also need to have exactly those attributes. We currently think about simplifying the creation of those data types, because if all attributes are fixed and needed they could be generated automatically and could be removed in the type definition.

The auditory component supports a second type of auditory signals that are handled a little different. You can assign `modality="auditory_speech"` which is used for a different model that processes spoken text. The current implementation for text comprehension does not continuously and incrementally update the memory after each word that is heard and understood. The text message that is send to CASCaS is written into the memory component after a certain processing time that is calculated dependend on the length of the text. The attributes of a speech message object are the same as those for the tone object plus an attribute named "text", which is of type string and contains the textual expression of the message. The current application for such messages is rather limited because so far the only CASCaS model where we needed speech was a pilot-/copilot model in the aeronautics domain. Most of the relevant communication were command like messages, very often with a defined syntax and terminology and very specific meanings. Also errors in the communication were not considered, so there was no need to integrate recognition, understanding and potential sources of errors that could happen. In future the cooperation aspect will be part of our modeling efforts which may lead to more sophisticated models for speech recognition. Obviously, speech is a learned skills, therefore a lot of learned declarative knowledge about a language and grammar should be included into a model for speech recognition.

**2.2.3 Haptic Perception**

So far no model for haptic perception is integrated into CASCaS. In general one can define objects and assign `modality="auditory_speech"`, but no detailed cognitive processes behind are modelled. Currently, those cues are directly written into the memory and can be assessed. The haptic input channel will be refined in the future.

**2.2.4 3D EGO-centric coordinate system**

To understand how CASCaS deals with object and the spatial layout it is important to know about the ego-centric coordinate system which CASCaS uses to represent the world around. This means that
the model assumes it is always located at the xyz position (0,0,0). This is important for all calculations of the visual field (which objects are within the field), but also the auditory signals are interpreted in this coordinate system. In general, all objects are represented in the ego-centric perspective and the coordinates of all objects are perceived and stored in the memory within this coordinate system.

Typically, driving or flight simulator software uses world coordinates to represent all objects. The transformation into an ego-centric coordination system is done by CASCaS internally, all that has to be done is to define a certain input object in the data definition file which represents the ego position in world coordinates. This object is used by the perception components to transform all other objects into the ego-centric system by subtracting the ego position from the coordinates of all other input objects.

Tip: An example about transformation of world coordinates into ego-centric coordinates at simulation time is given in the tutorial (section 3).

2.3 Goal Module & Associative Layer

These two components form the main execution unit that enable CASCaS to exhibit goal-oriented rule-based behavior. The goal module stores a number of intentions or goals that the model tries to fulfill (Goal: "What do I want to do next?"). Goals may be either short-term, which can be finished by doing some simple actions or they can consist of a lot of substeps (so called sub-goals) which contain a larger number of action sequences. Additionally, some goals may have to be executed periodically, for example a pilot has to monitor his instruments throughout the whole flight, though these intentions stay active over a longer period of time.

Each goal has a number of associated procedural action plans which specify necessary actions to fulfill the goal (at least partially). These action plans are stored in a rule-based format in the procedural memory component (Rule: "How can a certain goal be achieved?"). A rule consists of two parts: 1) the boolean precondition part typically specifies a situation where the rule shall be applied. If applicable the action part is executed and contains instructions like: shift the visual gaze to observe a certain display or to move the hands to manipulate some controls of the environment. To solve more complex tasks the model can use a number of successive rules and derived subgoals to gather information before it finally draws conclusions that it can memorize and use for decision making.

2.3.1 Goal Selection & Execution

During a simulation, the goal module can hold a number of active goals which have to be executed in "parallel", e.g. a pilot has to monitor lots of cockpit displays, check for up-to-date weather reports or adjust (reschedule) his flight plan according to unforeseen events. At each time of the simulation one of these goals is the so called "current" goal while others remain active in the background. The goal selection phase is the first step of the cognitive cycle. Afterwards, the associative layer requests the associated rules of the current goal from the procedural memory and selects one of them. This is the second step of the cognitive cycle. The third and final step is the execution of the rule's action side. The next example contains two simple rules which are used to explain how goals and subgoals are created and managed by CASCaS.

```plaintext
rule(goal=acquire_task){
  -->
  Goal(name=monitor_reaffic, mode=recurrent)
}

//attributes and items of a rule
rule(goal=monitor_traffic){
  Condition(boolean expression)
  -->
  Goal(observe_windshield)
  Goal(observe_left_mirror)
}
```

Each of the two rules is associated with a specific goal: the first rule is associated to acquire_task the second one to the goal monitor_traffic. The goal acquire_task is (by default) the first goal in the goal module if you launch a simulation and at least one rule is necessary for this goal, otherwise the model can not proceed. Besides the default goal you can define any number of goals and goal

5 The default goal name can be changed with a specific statement, see section Fehler! Verweisquelle konnte nicht gefunden werden..
hierarchies in a model that is necessary. It is important to understand, that goals are implicitly defined through the rules: 1) rules have a goal statement which associates them to a certain goalname. If the goal module selects a goal, all rules that are associated with this goal are potential candidates for selection and execution. 2) Rules can have a Goal statement in the action part. If the rule is selected and executed by the associative layer, this statement adds a new goal to the goal module.

The goal module contains a queue of goals which are executed one after another. Goals that are added to the module can have different parameters which changes the way they are processed:

- If a goal is added to the module with the additional parameter "mode=recurrent" the goal is executed periodically, which means that if the goal is the current goal and a rule is executed and fired, it is queued in again at the end of the goal queue.
- To remove such recurrent goals from the goal module an explicit GoalDone(goal name) statement on the right side of a rule is necessary (see section 2.5.2.3). The typical application for a recurrent goal are monitoring goals.
- Goals which are not declared recurrent are removed from the queue after a rule has been selected and executed.
- In some situations more than one rule is necessary before a goal should be changed. In this case an explicit GoalContinue (see section 2.5.2.3 about rule elements) statement can be written into the action part of a rule. If a rule with such a statement for the current goal is fired, it forces the goal module to do another cognitive cycle for that goal.
- The goals that are added to the module are queued first, which means they are executed immediately. It may happen that goals are pushed back and are executed late. To solve this issue that some goals are "lost" at the end of the queue there is the following solution ...
- If a goal is already put into the goal module and a rule wants to add the same goal again, no second instance is created, instead the existing goal is moved to the front of the queue and the next one that is selected.

Remark: in ACT-R the goal module does not automatically switch any goals. This behavior can be achieved by setting a special configuration parameter for the goal module: no_auto_goal_select. Possible configuration options are explained in the separat document: "Part2-Configuration File.docx".

2.3.2 Rule Selection & Execution

CASCaS subdivides between three different rule types regular, waiting and reactive. Each time a goal select is done, the associative layer requests a rule set from the procedural memory component for the current goal. In the following example three rules are given. The first two rules are already known from the previous subsection but the type attribute is added, the third rule is new and of type waiting. Furthermore, each rule has the necessary rule id attribute:

```plaintext
rule(id=1, goal=acquire_task, type=regular){
  -->
  Goal(name=monitor_traffic, mode=recurrent)
}
rule(id=2, goal=monitor_traffic, type=regular){
  Condition(boolean expression)
  -->
  Goal(observe_windshield)
  Goal(observe_left_mirror)
}
rule(id=3, goal=monitor_traffic, type=waiting){
  -->
}
```

Rule type regular

The associative layer always tries to select rules with type=regular for the current goal. If the conditional part of such a rule can be fulfilled successfully, it is selectable. The rule for the default goal acquire_task has no Condition which makes the rule selectable everytime when acquire_task is the current goal. Because there is only one regular rule with no precondition it is selected, executed and the goal monitor_traffic is added to the goal module as a recurrent goal.
In case that more than one regular rule is selectable at the same time a random uniform selection is done across those rules. The random selection can be configured by the addition of a weight parameter for each rule (see 2.5.2.1). The individual rule weights of each of the selectable rules are summed up and normalized for the random selection. After a regular rule is selected and its action part is executed, the processing of the current goal is successfully done and the associative layer sends a goal select request to the goal module which starts a new cognitive cycle. But what happens if no regular rule can be selected? In general, if no rule can be selected this produces a run-time error that aborts the simulation, because the goal-rule processing can not continue. But there may be situations where the goal can not be processed in a meaningful way, because nothing can be done at that specific point, the model simply may have to wait until a certain environmental state is reached.

**Rule type waiting**

To solve the situation when no regular rule can be selected, a rule with `type=waiting` should be provided in the rule set, which is shown in the example above for the goal monitor_traffic. Whenever no regular rule can be selected the associative tries to find a waiting rule for the current goal which it selects instead. The selection of a waiting rule automatically leads to a goal select request, but in this case the current goal is not removed from the goal module, because its execution could not be finished. The model assumes that it can process the goal later an it is queued again at the end of the goal module's goal queue. This rule type simplifies the specification of a rule-set which does not stall at run-time. If you have several regular rules with different conditions that fulfill the goal it can be very tricky or annoying to define the conditional statements that catch all opposing situations when no regular rule can be selected. Thus, waiting rules can be considered as the "fallback" solution, if 1) no real progress for the goal can be done at the current state or 2) the current state needs no actions that have to be done.

The selection and firing of a waiting rule is independent of the goal parameters, meaning that also a recurrent goal can (and should) have a waiting rule, because even if those goals are automatically queued again after the execution of a regular rule, they may also stall the goal-rule selection process if no waiting rule exists and no regular rule is selectable. As a rule of thumb one should always include at least one regular and one waiting rule per goal.

Waiting rules also accept all kinds of actions on the right side which can be used to model "default" actions that have to be taken if nothing else can be done. It is possible to write a GoalContinue statement into a waiting rule, which may be useful in a very small number of applications, but this can also very easily lead to a freeze of the goal / rule processing, because no new goal is selected if no regular rule can be selected.

**Remark:** waiting rule can even have a condition and you could also have several waiting rules for a rule. Multiple waiting rules in combination with conditions can be used to model special failure catching rules but it can be very tricky, so this is not advised if you are an experienced modeller, because conditions in waiting rules may lead to deadlocks in the goal-rule selection process.

**Rule type reactive**

Rules which use the type reactive are different from the previous two: they do not belong to a certain goal, hence the entry in the "goal" tag is ignored internally. Why is this type of rules necessarily needed? Certain events in the environment can appear at any time and cannot be considered as part of a certain action plan associated to a specific goal, e.g. within a traffic simulation consider a car that executes a brake action. A driver does not always explicitly wait for all cars around him to do a braking action, instead he reacts if such an event is perceived. Reactive rules can interrupt goal based-behavior to add new goals. The goals are added at the beginning of the goal queue (the default behavior for all new goals). In this case it assures that the interrupt can be handled adequately. It is upon the modeller to decide how many actions need to be done to handle the interrupt. Adding a goal with a separate rule set for interrupt handling offers the possibility to decide with different rules about the criticality of the interrupt. If it is critical, one can issue more goals or use even use a GoalContinue until the interrupt is handled adequately.

In the previous sections you have learned how object are defined, perceived and stored by CASCaS and how goals and rules are processed to process goal oriented behavior. The next section will introduce how CASCaS can interact with an external simulator environment using his motoric resources.

---

6 Technically, the GoalContinue statement within a rule does not prevent the goal select request, but it prevents the goal module from changing the current goal, thus it will select the same goal once more.
2.3.3 Recurrent Goals

This section explains a special mechanism which can be used to create recurrent goals which are executed successively and also periodically. Please read the explanations with care, because this mechanism is intended for some very specific situations.

```c
rule(id=2, goal=monitor_traffic, type=regular){
  Condition(boolean expression)
  -->
  Goal(observe_windshield, recurrency=50)
  Goal(observe_left_mirror, recurrency=5)
}
```

The parameter *recurrency* is optional and changes the characteristics of the goal selection. After the execution of a regular rule, goals which have this parameter are not removed from the goal module, hence they are either queued for later execution or they are successively selected again a number of times. The decision if a goal is selected successively or queued for later execution is a probabilistic decision after every goal execution step. The following short algorithm explains this procedure:

```c
//global variable of the goal module which counts the number of //successive executions of the currently selected goal. int number_of_executions=0;

//function called by the goal module after every goal execution void check_for_successive_execution(Goal goal){
  //increase variable by 1 after each goal execution number_of_executions++;

  //variable goal is the current goal: e.g. observe_windshield, //observe_left_mirror. goal.recurrency is the value in the //corresponding Goal() statement of a rule (see example above). int probability=goal.recurrency - number_of_executions;

  //random integer value [0 ... 100] int r=random(0, 100);

  if(r < probability){
    //the goal is selected / executed once more select_goal(goal);
  }
  else{
    //goal is queued in for later execution queue_goal(goal);
    //reset number, because another goal is selected now, therefore //number of executions for that goal is 0 initially. number_of_executions=0;
  }
}
```

Such recurrent goals have some characteristics: 1) the higher the recurrency value the higher the probability of multiple successive selections which implies a higher average dwell time. 2) The higher the number of successive selections, the lower the number of another successive selection and the higher the probability of a task switch. This is the result of the increasing `number_of_executions` which automatically reduces the probability value each cycle. 3) The recurrency value is not restricted to be <= 100, but if it is > 100 the goal will be executed a number of times successively without being interrupted. Warning: unreasonable high values may get the model stuck in that goal.

What is the intention behind the recurrency mechanism and does it explain dwell times in a cognitive plausible way? Several aspects have to be considered here:

---

7 algorithm is C-Code with comments in green color.
Part 1: An Introduction to CASCaS

1. **recurrency=0**: the goal is executed once and is queued in. In this case the mechanism is used to model periodic goals, but no successive executions are done. The dwell time is defined by the content of the rules only, no extra time is spend on that goal.

2. **recurrency > 0**: for a specific group of tasks this mechanism can be used as a plausible modeling concept for dwell time modeling: consider a situation where an operator has to scan (visually) several areas of interest (AOI). The scanning may be the one hand used to check certain information (which can then be modelled with a set of rules), but it may also be the case that he has to check the AOI repeatedly, because he has to wait for a certain important event that may appear. This reapeated checking and waiting for changes is learned task knowledge and must not be mixed up with cognitive concepts of *forgetting* values therefore one has to again look at a certain instrument. ACT-R uses the mechanism of *activation* which has nothing to do with this concept.

3. **recurrency > 0**: if the processes which lead to a certain dwell time for a task are not known and can not be modelled adequately with a number of rules, the repeated execution may be used to prevent the model to switch to other tasks more often or faster than humans actually do. In this case the model will stick with the goal and repeat certain actions or some "idle" rules could be modeled. This kind of modeling does not sufficiently explain the dwell time but at least prevents unrealistic task switching times which may ruin the complete model. This is not the preferred application, but sometimes the only possibility to have a model with adequate execution times for the overall set of tasks with the drawback that some dwell times can not be explained.

4. If **recurrency>0** is used it has to be checked that the distribution of the dwell time of human experimental data fits with the distribution of the given formula, otherwise the model will not be adequate.

5. The recurrency modifies the dwell time for successive executions, the "off" time (when the goal is not selected) can only be influenced by the number of additional goals which are on the agenda at the same time and selected in the mean time. Those other goals may also be defined using recurrency which can even increase the "off" time.

6. In contrast to a fully probabilistic goal selection across all goals which are active in the goal module, this implementation does not modify the selection order of goals, which makes the model more controllable but less flexible.

In general: because the concept could be used to just "fix" inadequate task dwell times without a cognitive adequate explanation behind it, one should be very careful with this statement and always explain why recurrency is used in a given model.

### 2.4 Autonomous Sensory and Motor Patterns

On the lowest level of processing the autonomous layer is used to model *highly learned behavior* which occurs if repetitive actions have been done over a long period of time. An example for such a behavior are steering actions while driving a car. When starting to learn driving the novice has to invest some effort to precisely steer the car, he has to learn how much steering is necessary to force a certain reaction of the car. Also, if you change from your private car to another one, the behavior is often at least a little different, so you have to adjust. With enough training, the fine control of your muscle movement for the steering gets very sophisticated, the steering gets very precise and less conscious intervention is necessary to correct the small deviations. Steering a car is only an example, typical actions that people have learned are pressing knobs on various devices, dialing in values using a potentiometer, using levers, keyboards or other input and control devices - in general, everything that can be manipulated in the environment uses *sensory-motor patterns* for muscle movement which get more and more sophisticated with learning.

We also apply learned knowledge in new situations: for instance a user manual of a new electronic device explains the functionality for all the knobs of the remote control. In general, we can use the remote control but we have to remember the layout and the meaning of some special knobs. A numeric keyboard with numbers from 0-9 is similar on many devices and can be used "intuitively" because it is well known.

---

8 The optimization process to achieve such highly learned behavior is not modelled in CASCaS.
In the CASCaS, the learning of new declarative knowledge (the layout of a remote control or the function of a certain key) is not implemented, therefore declarative knowledge about controls that the model should use in a simulation must be added as already learned knowledge facts. The actual press of a button will always trigger a motor action "press the knob" on the autonomous layer. The sensory-motor program can be associated with the button that should be pressed.

To manipulate the environment we extended the procedure language with specialised symbolic actions to use combined arm and hand movements, which can also be found in other cognitive architectures, like ACT-R/PM (Anderson et al., 2004) EPIC (Kieras and Meyer, 1997) or APEX (Freed, 1998). Those actions consume time, dependent on the length of the movement to grab or reach a certain manipulator and also the complexity that is necessary to adjust the wanted position / value.

The following actions have been implemented:

- **Move**: move the hand to the location of an object
- **Unguided move**: unguided move to the resource (without the visual feedback of the eyes).
- **Grasp**: grasp the object, if resource not already moved to this instrument, move action is automatically done
- **Release**: release the object (precondition: grasped object before)
- **Adjust**: adjust the object to a new value (e.g. dial in a value in a potentiometer, shifting the gear, steer the wheel, ...)
- **Type**: type in a word or value into a keyboard (like AOI (a grasp of the keyboard has to be executed before)
- **Mouse-move**: move the mouse to a new location (a grasp on the mouse has to be executed before)
- **Mouse-Click**: click with a mouse on a location (precondition: mouse grasped)
- **Mouse-double-click**: click with a mouse twice on a location (precondition: mouse grasped)
- **Push**: push a physical button
- **Pull**: a physical object (e.g. altitude selector, direction indicator)

For the actions release, adjust, type, mouse-move, mouse-click, push, pull the resource has to be grasped before. A grasp may include a movement.

The following rule contains a **Motor** statement which demands the right hand resource. The manipulation that the model should do is to adjust the lever to the value 1. The example contains the data type for a lever object and the definition of an output channel:

```
rule(id=1, goal=use_lever, type=regular){
  Motor(right hand, adjust, instr=lever.position, value=1)
}
```

```
<tns:type name="lever" modality="visual">
  <tns:variable name="id" type="int" />
  <tns:variable name="x" type="double" />
  <tns:variable name="y" type="double" />
  <tns:variable name="z" type="double" />
  <tns:variable name="position" type="int" />
</tns:type>

<variable_definition>
  <variable name="leverobject" type="lever" mode="out">
    <options> </options>
  </variable>
</variable_definition>
```

At the end of this section the main modules have been introduced that are used for sensing input data and creating output actions with the rule base procedure language. So far we have not cared about the memory component and we also have not introduced all statements of the procedure language. The details of the memory component are not part of this introductory documentation, because you can start to build some models without a detailed understanding of concepts like activation for example. What you need to know is the complete syntax of the procedure and the other necessary
configuration files that are required to set up a simulation. This will be the content of the following section of this documentation.

2.5 CASCaS Procedure Language

The procedure language is used to create a procedure file (file ending ".proc") which is a simple text file in ANSI text format. The procedure file is necessarily needed and does not only contain the rules (section 2.5.2) for the task that should be simulated. It can also include additional declarative (or factual) knowledge (section 2.5.1), a number of initialisation statements for different modules (section 2.5.3) and finally you can use #define statements which are similar to the statement of the C-preprocessor (section 2.5.4). These four parts of the procedure language are introduced in the following subsections. We advice to use Notepad++ as texteditor which allows to save textfiles in this format. A syntax highlighting file can be downloaded from our website for Notepad++, which will show you highlighted keywords, strings, comments and numbers as known from programming / script lanugages. Besides the procedure the data descrition file (.xtop) has to be written. This file is best viewed as .xml document.

**IMPORTANT:** All input files for CASCaS must be stored as ASCII textfiles. Do not copy text from Word or pdf documents, because those files use a larger character set (e.g. UTF-8). We have experienced problems with e.g. ‘ ‘ signs in mathematical expressions, that were copied from word documents, and showed up as ‘ ‘ in Notepad++ but what actually has happened was that Notepad++ had automatically changed the filetype to "UTF-8 Without BOM" while pasting the text. Converting the file back to ANSI text showed the supposedly ‘ ‘ signs was a ‘ â€“ ‘ which obviously is not a ‘ ‘. So be warned, If you copy / paste from other documents: make sure that you have a look at the file encoding in Notepad++ and if it is no longer ANSI text after copy / paste, switch it back. If you now see strange symbols or something else that is not as wanted, replace those characters. Probably the best solution is: never copy anything from a file which is not a ANSI textfile: word, .pdf, html websides, emails ...

2.5.1 Specification of declarative knowledge

Declarative knowledge is factual knowledge that people have learned during their lives for example about other persons or objects - their appearance, functionality and also about relations between them. We have images in our mind from a lot of friends and family members, objects like our homes, work places, colleagues and so on. Also everyone should have learned a lot of stuff in school including languages, math and so on. Declarative knowledge also contains abstract knowledge like traffic rules, laws or other regulations that are specific to certain task domains or work environments. Car drivers (should) know the traffic rules and they also need knowledge about the functionality and technique of a car (how a gearbox works, a clutch and so on). All this is declarative knowledge and is used in our daily life.

In CASCaS declarative knowledge is stored in the corresponding declarative memory component. The structure of this component is a semantic network, as it can be seen in Figure 3. The network has a unique root node, which can be seen on the left side of the figure. Each arrow represents a named associative link between two nodes. Nodes can have a value (squares) or not (black cirles). In this example only the leave nodes have values (besides the root node, but this can be ignored here). In general, every node can have a value and any number of incoming and outgoing associative links.

The figure shows some knowledge facts that could possibly be used for a pilot model: the model has some knowledge about a number of airports in New York city and for

![Figure 3: Screenshot of CASCaS' memory visualisation showing some declarative knowledge facts.](image-url)
Part 1: An Introduction to CASCaS

each airport the name and a unique airport code are associated as attributes. How can this knowledge be used during a simulation? Assume the pilot (resp. the model) has to change the destination airport during flight, for example the instruction to approach New York is received during the flight. The model can remember three airports from New York: Newark, JFK and La Guardia and it can use his knowledge about the identifiers to find the positions on the map and to check which one he wants to use as alternative airport. This is a very simple example and typically we associate a lot of information with certain objects.

Declarative knowledge that a model should have for a specific simulation can be specified and loaded into the architecture before the simulation\(^9\). The code example below shows an initialization block for the memory component with a number of declarative statements. The declarative statement can use \textit{symbolic tags} \(<\text{unique tagname}>\) which refer to a specific node element of the network that can be reused in later statements. Tags are used and valid only during the initialization, they are not stored in the memory component itself. Figure 3 shows three \textit{airport} links: if another link (\textit{f.e. code}) should be appended after one specific airport link this is only possible if you can refer those airport links directly. Hence, the unique symbolic tags are necessarily needed.

```csharp
memory{
  declarative(variable=<a>:city)
  declarative(variable=<a>.name, value='New York')

  //insert an airport with its name and code
  declarative(variable=<b>:<a>.airport)
  declarative(variable=<b>.name, value='Newark')
  declarative(variable=<b>.code, value='EWR')

  declarative(variable=<c>:<a>.airport)
  declarative(variable=<c>.name, value=La Guardia)
  declarative(variable=<c>.code, value='LGA')
  //further statements to declare the other airport
}
```

The code snippet above shows the necessary instructions in the procedure file which are needed to create the small network of declarative knowledge:

7. The first associative link \textit{city} is tagged with the symbolic name \(<a>\) .

8. The second declarative statement uses this tag and appends a further link \textit{name}, with an additional value 'New York' which is stored in a node. The dot notation \(<a>.name\) is known from programming languages and is interpreted by CASCaS to create chains of links. It is also possible to append multiple links within one declaration. A number of successive associations which point to a specific node are called \textit{memory path}.

9. The third declarative statement reuses the tag \(<a>\) and creates appends the link \textit{airport} which leads to the memory path \textit{city.airport}. The symbolic tag \(<b>\) is assigned to this path. In the successive statements \(<b>\) is reused to attach the \textit{name} and the \textit{code} of the Newark airport. The following three lines show that the next airport is defined in the same way but it uses another unique tag \(<c>\) instead of \(<b>\).

In section 2.2.2 (auditory perception) an example was given which referred to two different input channels for auditory signals. It was also remarked that the channel names have no meaning for the cognitive model and that the model has to use additional declarative knowledge to decide if a signal is emitted from different objects: the left speaker or the right speaker. The example contained two channels for auditory tones which could transmit signals in parallel. Currently, the model has no sophisticated spatial memory concept but this is one part of our research and will be integrated within the next releases. One possibility to model the relation between an auditory cue and a speaker uses the following declarative knowledge:

```csharp
memory{
  declarative(variable=<a>:speaker)
}
```

\(^9\) Only models for very simple tasks that are highly proceduralised may not use any declarative knowledge.
The specified declarative knowledge contains two speakers left and right and attaches a spatial coordinate. The coordinate system if OpenGL (forward: -z, right:+x, up:+y). Furthermore we need the data type and the channel definitions which are taken from section 2.2.2:

```xml
<tns:type name="Audiosignal" modality="auditory_tone">
  <tns:variable name="id" type="int" />
  <tns:variable name="x" type="double" />
  <tns:variable name="y" type="double" />
  <tns:variable name="z" type="double" />
  <tns:variable name="frequency" type="double" />
  <tns:variable name="amplitude" type="double" />
</tns:type>

<variable_definition>
  <variable name="speaker_1" type="Audiosignal" mode="in">
    <options> </options>
  </variable>
  <variable name="speaker_2" type="Audiosignal" mode="in">
    <options> </options>
  </variable>
</variable_definition>
```

A rule set of two rules is necessary to implement a very simple spatial recognition for audiosignals. It is assumed that the height of and longitudinal distance to both speakers is the same. The first rule classifies signals from the left side, the second rule from the right side.

```java
rule(id=1, goal=use_lever, type=reactive){
  Condition(audiosignal.attended == 'no')
  Condition(audiosignal.x == speaker.left.x)
  -->
  Memorize(audiosignal.direction, cue_left)
}
rule(id=2, goal=use_lever, type=reactive){
  Condition(audiosignal.attended == 'no')
  Condition(audiosignal.x == speaker.right.x)
  -->
  Memorize(audiosignal.direction, cue_right)
}
```

### 2.5.2 Rule specification

This section contains the syntactic elements of the procedure language. It starts with the basic parts of a rule, which consists of 1) the rules’ head with some configuration attributes, 2) a conditional part: the search pattern (what is also called the left hand side of the rule, short: lhs) and 3) an action pattern (also called right hand side, short: rhs). If the search pattern can be successfully found (retrieved) from the memory component, the rule can be selected. If multiple rules for a goal can be selected at the same time the model randomly selects one (see explanation for weight below 2.5.2.1). The rule that is finally selected from the conflict set is executed, meaning the model will initiate the actions of the action pattern. The search and action pattern are separated by the specific symbol "-->".

```java
//components of a rule
rule(head){
```
Part 1: An Introduction to CASCaS

```plaintext
LHS // search pattern
--> // rule side separator
RHS // action pattern
}
```

The next code snippet shows some more details what can be specified in the three different parts of a rule. All of them are explained in the following paragraphs.

```plaintext
//attributes and items of a rule
rule(id=int, goal=name, type={regular|reactive|waiting}, weight=float){
  Retrieve(object, age)
  Condition(boolean expression)
  -->
  LookAt(input object)
  Motor(output object, value)
  Memorize(object, value)
  Associate(object, link name)
  Goal(goal name)
  GoalDone(goal name)
  GoalContinue
}
```

### 2.5.2.1 Rule Head

```plaintext
rule(id=int, goal=name, type={regular|reactive|waiting}, weight=float)
```

The rule head contains four basic parameter which are necessary for each rule.

**id**

Each rule needs a unique integer number. It is necessary and the uniqueness is checked when the procedure file is loaded by the architecture. It does not have any impact on the modeling itself, it is just needed for technical reasons. For debugging and analysis purposes selected rules (resp. the id's) are written in the simulation trace.

**type**

The rule type can be either regular, waiting or reactive. If no regular rule can be selected a waiting rule is chosen. If no rule can be selected the cognitive cycle will throw an error and can not continue the simulation. Unlike to regular and waiting rules, reactive rules are not associated to a goal and can fire during the execution of any goal. They are used to model reactive behavior. See section 2.3.2 for more information about the execution of rules.

**goal**

Rules with either type=regular or type=waiting must be associated with a certain goal name. Besides the search pattern of the lhs the goal name can be considered as an additional precondition for the rule.

**weight**

The weight is a floating point value between [0...1] which is used for the probabilistic rule selection if more than one rule of the same type (either regular or waiting) are selectable (their search patterns could be found / matched in the memory component). Because only one rule can be selected a probabilistic rule selection is done across the conflicting rules. All rules from the same type (regular or waiting) are put into a conflict set. The associative layer first checks the existence of regular rules in the conflict set, if none exists it checks the waiting rules. If again none exists the selection throws an error and the simulation is terminated. For each of these steps the weights of the rules (regular or waiting) are summed up and each individual weight is divided by the sum: this process is called normalization of rule weights. The normalized weight of each rule is the probability for its selection. To select one of the rules the associative layer draws from a random uniform distribution.
2.5.2.2 LHS: Items of the search pattern

The left hand side of a rule can be considered as a search pattern across the models memory. If the pattern which consists of Retrieve and Condition statements can be matched, the rule is selectable and is added to the conflict set.

Retrieve(memory path, age)

The retrieval request searches the memory for the occurrence of the specified memory path. Section 2.5.1 explained what a memory path is: a chain of associative links which point to a specific node, typically an object or an attribute of an object. The age parameter specifies that the node specified by the Retrieve command must not be older than the given age value. Older means that the last time this node was written by the perception or by an explicit Memorize (see below, RHS items, 2.5.2.3) statement was not before current simulation minus age in milliseconds. This has nothing to do with remembering and forgetting, it is additional knowledge, that certain information has to be “up-to-date”.

- If age is set to -1, the age does not matter, can be matched. If the model can remember a value it can be used.
- Any other numeric value > 0 is considered as a threshold in milliseconds.

If a goal contains regular rules with retrieval statements that require time-critical information, one has to consider that those rules are not selectable if the information is out-dated. If no regular rule is selectable a waiting rule is chosen instead but in most cases at least one additional regular rule should complement the rule set for this goal which contains a LookAt command (next section). This Command moves the gaze towards the object and updates the required information in the memory component.

Condition(boolean expression)

The condition statement checks a boolean expression. If the condition can not be evaluated to "true" the rule will not be selectable. The boolean expression can contain any number of checks across a number of memory path elements. The possible operators that can be used within the expressions are:

- &&, ||
- <, >, <=, >=, !=,
- +, -, *, /,
- (, )

The elements that can be compared are

- memory paths (which refer to concrete values). Nodes without a value can not be compared.
- numbers (floating point and integer)
- String variables can contain any number of the following chars: a-z, A-Z, 0-9, _, - and they are embraced by single quotes ' , e.g.: ‘hello_123’, ‘yes’...

2.5.2.3 RHS: Items of the action pattern

If the left hand side was matched successfully and it was selected by the associative layer the right hand side (action pattern) is executed. The action pattern can contain a number of commands which trigger motoric actions move the visual perception or they are used to actively memorize certain values or to add certain relations (associative links) between nodes:

LookAt(memory path)

Motor(resource, type, memory path, value)
Memorize(memory path, value)
Associate(memory path, memory path)
Goal(goal name)
GoalDone(goal name)
GoalContinue

It is important to know that commands which use memory nodes represented as by a memory path have to be matched on the left side, either with an explicit Retrieve statement or implicitly if the memory path is used (and automatically matched) in a Condition statement. This means if one of the following commands is used in the action pattern:

- add a value (Memorize) to an existing memory path
- adding an associative link (Associate) which points to an existing memory path
- shift gaze to an object (LookAt)
- manipulate an object (Motor)

the memory path to the corresponding node must be retrieved in the search pattern.

Important: The general mechanism is that the search pattern has to retrieve those memory nodes, that are used in the action part commands.

LookAt(memory path)

The model shifts its visual attention towards a certain object object using a coordinated head / eye movement. The object is specified through the parameter memory path and must point to the base node of a defined type. In section 2.2.1 a car object was used to explain how the visual perception stores objects in the memory component and the term base node was defined.

```plaintext
rule(id=1, goal=observe_cars, type=regular, weight=1.0){
  Retrieve(car, -1)
  Condition(car.attended == 'no')
  -->
  LookAt(car)
}
```

This rule retrieves any car object (age check -1) which was not attended previously. If such a memory path can be retrieved and the condition is also true, the matched memory path can be used as input to the visual perception.

Motor(resource, type, memory path, value, guidance)

The model initiates interaction with the environment, e.g. pressing a knob, dialing in some values or using a steering wheel and the pedals of a car. A motor command always triggers a sensory-motor pattern on the autonomous layer which may run in parallel to the associative layer. Interference with other tasks may occur if the required input is not available because the models visual attention is not targeted towards the necessary information sources. Attentional distraction due to task interleaving are currently not part of the model.

10. resource can be either
   o left hand or right hand.

11. type can be:
   o Move: move the hand to the location of an object
   o Unguided_move: unguided move to the resource (without the visual feedback of the eyes).
   o Grasp: grasp the object, if resource not already moved to this instrument, move action is automatically done
   o Release: release the object (precondition: grasped object before)
   o Adjust: adjust the object to a new value (e.g. dial in a value in a potentiometer, shifting the gear, steer the wheel, ...)
   o Type: type in a word or value into a keyboard (like) AOI (a grasp of the keyboard has to be executed before)
Mouse-move: move the mouse to a new location (a grasp on the mouse has to be executed before)
Mouse-Click: click with a mouse on a location (precondition: mouse grasped)
Mouse-double-click: click with a mouse twice on a location (precondition: mouse grasped)
Push: push a physical button
Pull: a physical object (e.g. altitude selector, direction indicator)

12. *memory path* can be a path to a base node of a defined type (similar to the LookAt command). For objects of those type at least one output channel must be defined, otherwise the Motor command will throw an exception. An example for a motor command which includes an output channel and a type definition was already given in section 2.4 and is shown below.

13. *value* depends on the defined type and the memory path element that is referred to. If the memory path points to an integer data type, e.g. `lever.position`, value must contain an integer number.

```
rule (id=1, goal=use_lever, type=regular){
  -->
  Motor (right hand, adjust, instr=lever.position, value=1)
}
```

```
<tns:type name="lever" modality="visual">
  <tns:variable name="id" type="int" />
  <tns:variable name="x" type="double" />
  <tns:variable name="y" type="double" />
  <tns:variable name="z" type="double" />
  <tns:variable name="position" type="int" />
</tns:type>

<variable_definition>
  <variable name="leverobject" type="lever" mode="out">
    <options> </options>
  </variable>
</variable_definition>
```

Memorize(*memory path*, *value*)

The model can explicitly memorize additional information which it concludes from the current situation. These conclusions may be remembered (Retrieve) within the search pattern of any other rule. If the parameter *memory path*:

14. does not exist, a new path is added as specified and the value is appended as destination node of the path.
15. was partially matched in the search pattern, all path elements that were not matched are created and value is appended as destination node of the path.
16. is fully matched, a new value is appended to the path.

Associate(*new memory path*, *target memory path*)

Any object of the declarative memory can be targeted by any number of additional associative links. Within a certain situational dependent context an object may have a specific meanings which can be expressed by an associative link.

For example the model approaches a car and at a certain point the model considers the car a "lead car". Associations are an important part of the internal situation representation of the model.

The parameter *target memory path* must be fully matched in the search pattern. *new memory path* can be partially matched until the last path element, but must not be fully matched. If the full path is matched, no new association can be added. If all or less than the last path element could be matched, the missing elements of the path are created.
Part 1: An Introduction to CASCaS

**Goal(name=goal name, recurrency=value)**

The model may generate a new intention (goal) which is put into the goal modules agenda. New goals are always selected and executed immediately.

- Parameter *name* refers to the name of a goal. Rules which refer to this name in their rule *head* are associated to this goal and are part of the rule selection process for this goal.
- Parameter *recurrency* is optional. After the execution of a regular rule (see section 2.3.1 about goal selection and execution), the goal is not finished, it is either queued for later execution or it is successively selected again. The decision if the goal is scheduled again later or successively is a probabilistic decision and it is influenced by the value of *recurrency*:

\[
P(g) = g.\text{recurrency} - \text{number\_of\_executions}
\]

- The decision if a goal is executed successively or if it is selected again successively or if it is queued for later execution is calculated after every goal execution step:

```java
int number_of_executions=0; // globally defined
//function called by the goal module after every goal execution
boolean check_for_successive_execution(){
    int r=random(0, 100);
    number_of_executions++;
    if(r < P(g)){
        execute(g);
        return true;
    }
    else{
        queue(g);
        number_of_executions=0;
        return false;
    }
}
```

- Recurrent goals have some simple characteristics: 1) the higher the recurrency value the higher the probability of successive selection. 2) The higher the number of successive selections, the lower the number of another successive selection.
- How can this be applied?
  - Example: if the model has to scan several areas of interest (AOI), and one area has a chance that critical events may appear, but it is not known when or how often, recurrency should be high so the successive dwell time on that AOI is higher. But the number of other goals is also important, because when the goal is queued, other goals also have a chance that they are executed.
- Drawback:
  - the mechanism was implemented because for several tasks one may not be able to find out what processes lead to a certain long dwell time. Often one can assume some of the information that are gathered, but sometimes the dwell time on an AOI is that high because there is a chance that something interesting will happen. Therefore one waits for these events. To model dwell times for such situations this mechanism can be used.
  - It is not a general mechanism to generate cognitive plausible explanations about swell times.
  - Can not be used to explain the phenomenon of cognitive fixation, because the probability will be 0 sometimes. Can be used to model / simulate different gaze times.
- Recurrent goals can only be terminated with an explicit *GoalDone* statement (see next).

**GoalDone(goal name)**

This statement can be used to terminate a goal immediately. If this item is fired, the goal module searches for the existence of this goal and the goal is removed. This statement results in a recursive descent through all subgoals which are also removed from the module. Important: This statement is the only possibility to terminate recurrent goals.
Goal Continue

The current active goal will remain first goal modules queue and therefore it is selected once more in the next goal selection step. With this command done can force the model continue the execution of the current goal if certain situation specific conditions need to handle this goal with a high priority and do not allow any interruption. For example if a driver model is about to leave the road it will need to steer back with very high priority, until it is on a safe course on the road again.

2.5.3 Basic Module Initialisation Statements

Each module can have some specific initialisation statements which may be extended in future version of CASCaS. These statements are encapsulated by a block that is named as its corresponding components are, e.g. memory, goal_module. The most important initialisation block was already introduced: it is the memory initialisation with the declarative knowledge commands. There are at least two more initialisations that are relevant for many models:

It is necessary to set an initial gaze direction for the visual perception. This is done in the motor component, because the movement of the eye and head is realised in the motoric section of the model. The parameter of the eye_start command has to be a valid base node of a specified type, but it can also be any area of interest which is specified in the topology file.

```
  motor{
    eye_start(speedometer)
  }
```

The second parameter that is used often is to set another default goal which to start the simulation. If this command is not set, the model assumes that the goal „acquire_task“ is the first goal and there must be at least one regular rule for that goal. If another goal should be used instead of acquire_task the following initialisation block is necessary:

```
  goal_module{
    default_goal(goalname)
  }
```

There are some more specific initialisation commands also for other modules but they are not discussed in this introductory documentation.

2.5.4 Preprocessor Statements

```define```

The `#define` command can be used very similar to the C-Code preprocessor command. It is used for text replacement and defines a search pattern which is replaced by a replacement pattern throughout the whole procedure file:

```
  • #define(search_string:replace_string)
```

Typically the defines are written at the very beginning of the procedure file. They search_string can be used within every command. It can also replace a complete conditional expression or a specific condition part which is often reused. A very good example are parameters of a model (e.g. numerical values) which are spread around the whole procedure. If you want to change one of those parameters you would have to search all the rules and replace the numbers with the problem that one may simply overlook some of them. If you define constants instead at the beginning of the procedure you can easily cluster the relevant model parameters and you can simply change the value by changing the replace_string.

If you connect CASCaS to a simulator you may often find numeric values which define certain environmental attributes (enum typed values). Also string data types are rarely transmitted via any sockets, often enum types are used instead You can use the define command to use names in your procedure which are replaced by the numeric values transmitted by the simulator.

`#include “relative path to include another procedure file”`

For some larger models it is useful to split the procedure file to separate rules or also the declarative knowledge specification into different files. The `#include` command allows to include additional
procedure files into each other. This is helpful in many ways: a number of rules can be separated into a different file and can be reused in different models. This allows to create reusable modules of rules. Another application is to have init statements or defines that should be included in every model.

### 2.5.5 Example Procedure File

The procedure file content was explained in detail in the previous section. To create a new procedure file, create an ASCII text file with the file ending ".proc". The syntax of the procedure file allows single-line comments (//) and multi-line comments (/* text */). The initialisation statements and the rules can be mixed, but this is not recommended because readability will suffer. The init statements should be written first, the rules afterwards. It is intended that each component can have an initialisation section, but up to now not all components do have specific commands and parameters that can be set. This section lists the most common parts which can be found in a procedure file and is a guideline how the files could be structured:

```plaintext
/\ REQUIRED: the version tag has to be defined first in every procedure
version(1.1)
/\ a define which is used later on. eyes.focus is generated internally
/\ by the visual perception and always associates the base node of the
/\ object that is currently focused. The isa attribute is added by the
/\ visual perception to each object and corresponds to the string
/\ representation of the objects defined type (name of the type)
#define(VIEWPOINT:eyes.focus.isa)
/\... more defines

/\ possible include files with a path relative to this file
#include "\modules\test.proc"

/\ OPTIONAL: The initialisation of the declarative memory was
/\ explained in section 2.5.
memory{
  declarative(variable=<a>:city)
  declarative(variable=<a>.name, value='New York')
  // insert an airport with its name and code
  declarative(variable=<b>:<a>.airport)
  declarative(variable=<b>.name, value='Newark')
  declarative(variable=<b>.code, value='EWR')
}

/\ REQUIRED: Initialisation of the motor components gaze
motor{
  // The only statement that is used here at the moment is to
  // initialise the gaze direction of the eye. The instrument
  // has to be a valid object, by default it should be the root
  // object of the topology file (see next section)
  eye_start(root);
}

/\ OPTIONAL: The only statement that can be given to the goal module as
/\ initialization parameter is the first goal that should be processed
goal_module{
  default_goal(golname)
}

//Define all rules for the procedural knowledge base\\

rule(id=1, goal=observe_speedometer, type=regular){
  Retrieve(car, -1)
  Condition(VIEWPOINT != 'car')
  -->
  LookAt(car)
  GoalContinue
}
```
With these final explanations the basics to set up a model in CASCaS are introduced. The sections gave an overview of the most important components which are needed to understand the goal oriented, rule based procedure execution and states the basic knowledge about perception and action. The last and final chapter wraps up all the specifications and explains the necessary files that have to be written to set up a simulation.

3 CASCaS Simulation Tutorial

In this section a minimal simulation setup is explained and the processing of data is shown step by step. First, an overview of the CASCaS installation directory and the necessary files and locations are given, afterwards the data definition and the configuration file which are used in this tutorial are explained. The startup utility to configure a simulation and the CASCaS Event Scheduler which controls the simulation are introduced. Finally, the procedure file is incrementally changed to explain the processing of rules step by step. The directory structure of a CASCaS installation contains a number of subdirectories and files:

- base directory
  - config
    - dns
    - procedures
    - test_cases
  - etc
  - lib
  - tmp
  - tools

The base directory contains all the main executable cascas.exe and all additional binary files (most of them are dynamic link libraries, short .dll). CASCaS is written in Plain C-Code and uses two external programming libraries: the glib which contains basic data structures and GTK+ as a window toolkit. Most of the dll's that are found in the base directory are part of those libraries. The subdirectories etc and lib can be ignored right now, they contain some specific GTK+ binaries and configuration files.

The most important directory for all the modeling work is the config directory. This directory contains the configuration files: each configuration file specifies the necessary input files and parameter for a simulation (explained in section 3.2). The most important subdirectory is procedures which contains the procedure files (.proc) and also the data definition files (.xtop). The dns directory is important for the socket scheduler environment only and its content is explained in the document Part3-SocketScheduler.pdf. The next section starts with an example data definition file that is used in this tutorial.

3.1 Data Definition File

The data definition file is an .xml file that, contains three sections: the type definition and the input output channel definition (section 2.1.1). Additionally, a topology definition section to define static environment elements is defined, but the details of this section go far beyond this document. So far it is sufficient to know that the topology file can be used to define static environment elements, that do not move during the simulation. This is true for example, if a pilot model sits in a cockpit. In this case the cockpit can be modeled in the static topology but as soon as an object moves during the simulation it can not be modeled in the static topology, it has to be defined as an object which has to be received through a data channel. The advantage of a static definition is the possibility for a pre-computation of eye movement paths between all objects which are defined in the section. This saves some computation time during the simulation which may be useful for a large number of objects. For this tutorial a deeper understanding of the topology section is not relevant, only the defined root node

Page 26
<tns:root_aoi name="root"> is used to initialize the gaze direction. The example below can also be used as template:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tns:environment version="1.0" xmlns:tns="http://www.offis.de/xenv"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.offis.de/topology ../../../src/topology/xenv.xsd">
  <!-- type definition, see section 2.1.1 -->
  <tns:types>
    <tns:type name="Car" type="dynamic" modality="visual">
      <tns:variable name="id" type="int"/>
      <tns:variable name="x" type="double"/>
      <tns:variable name="y" type="double"/>
      <tns:variable name="z" type="double"/>
    </tns:type>
  </tns:types>

  <!-- variable instantiation, see section 2.1.1 -->
  <variable_definition>
    <variable name="car_1" type="Car" mode="in">
      <options> </options>
    </variable>
  </variable_definition>

  <!-- static environment definition: explained in a separate document, you may use the default definition below for many models. -->
  <tns:topology>
    <tns:fov_y>60.0</tns:fov_y>
    <tns:z_near>1.0</tns:z_near>
    <tns:z_far>10000</tns:z_far>
    <tns:infopanel>true</tns:infopanel>
    <tns:default_light> <!-- light node -->
      <tns:light_ambient x="0.2" y="0.2" z="0.2" w="1.0"/>
      <tns:light_diffuse x="0.7" y="0.7" z="0.7" w="1.0"/>
      <tns:light_specular x="0.7" y="0.7" z="0.7" w="1.0"/>
      <tns:light_position x="-0.5" y="0.0" z="0.0" w="1.0"/>
    </tns:default_light>
    <tns:default_shader>def_light</tns:default_shader>
    <tns:translation_coord x="0.0" y="0.0" z="0.0"/>
    <tns:look_at_default x="0.0" y="0.0" z="-1.0"/>
    <tns:head_pos x="0.0" y="0.0" z="0.0"/>
    <tns:head_rot x="0.0" y="0.0" z="0.0"/>
    <tns:left_hand_idle_pos x="0.0" y="0.0" z="0.0"/>
    <tns:right_hand_idle_pos x="0.0" y="0.0" z="0.0"/>
    <tns:left_foot_idle_pos x="0.0" y="0.0" z="0.0"/>
    <tns:right_foot_idle_pos x="0.0" y="0.0" z="0.0"/>
  </tns:topology>

  <!-- This node defines the top level aoi. -->
  <tns:root_aoi name="root">
    <tns:shape name="shape_root">
      <tns:cuboid group="false">
        <tns:centrum x="0" y="0" z="-1"/>
        <tns:size d="1" h="1" w="1"/></tns:size>
        <tns:filled>true</tns:filled>
        <tns:path_to_texture/></tns:path_to_texture>
      </tns:cuboid>
    </tns:shape>
  </tns:root_aoi>
</tns:environment>
```
This data definition file is used as basis for the next steps in the tutorial.

### 3.2 Configuration file

The configuration file is the top level file that specifies all necessary input for a simulation. The configuration file can be written manually with a text editor, but it is advised to use the utility program **CASCaS_Startup.jar** which can be found in the base directory of the CASCaS installation. It loads all configurations files that are stored in the `config` directory.

The screenshot of the startup utility shows the clean installation which contains the tutorial configuration file called `part_1_doc.cfg` with some basic parameters for the modules. In the top left you can select the configuration file. Afterwards all parameters will be filled into the different GUI elements. Top left you also see the buttons to select the executable (cascas.exe), to start the simulation and the third button "Update from Files" is useful, if you have manually edited a configuration file, while the startup utility is open. It reloads the file into its internal memory.
In the top right you see the files that are necessary for a simulation: you can choose the procedure and the data definition/topology file. Debug options should always be enabled, and you can reduce the number of debug output by decrementing the numeric value. Starting with CASCaS you should use the full output, which is very large, but it traces the whole simulation and all internal processing steps.

In the bottom left the parameter sets for the cognitive components of CASCaS can be modified. For this tutorial it is not necessary to care about all the parameters of the configuration. Currently the "output" module tab is selected: this module produces simulation traces which are stored in comma separated value files. They can be used for statistical evaluation of the simulation runs. If you want to edit one of the given values, click it and the entry will appear in the text fields below. For example click on "base_directory" and change the value to ".my_simulation". Afterwards click "Accept Changes" and the entry is modified. If you want to add further parameter, add name and value and click "New Parameter". If you want to delete a parameter, select one and press "Delete Parameter". If you want to save the changes to the configuration use the File menu and click on "Save .cfg" or "Save as .cfg".

Up to this point we do not care about all the other information that the utility shows, they will be explained in a separate document.

### 3.3 Simulation Environment Event Scheduler

This section deals with the connection of CASCaS to external simulator (environment) programs. CASCaS has two basic environments in the installation. The Socket Scheduler is a fully scriptable multi-socket environment which can use TCP or UDP connections to communicate with a number of different external programs at the same time. This document focuses on the Event Scheduler which is a step based simulator that can be used to program small experiments or to develop and test procedures before an actual realtime simulator is connected using the Socket Simulator. The figure below shows the event scheduler window if you select and start part_1_doc.cfg from the startup utility. The Event Scheduler offers different views: in this example the Memory visualisation button was pressed and the Memory tab was selected in the middle view.

To the left you can see the simulation controls, for example the Stop button to end a simulation or the Step button which simulates one simulation step (50ms). Besides the Step button the actual simulation step is shown. The text field and the slider below can be used to increase the number of steps that is simulated with each press of the Step button. The field below shows all input variables that can be manipulated during the simulation by inserting adequate numbers in the textfield (need to press "Enter" in every textfield). The data definition file that is loaded by the configuration contains a single data type with id,x,y,z as the only attributes and one input data channel named car_1. Therefore the values for car_1.id/x/y/z can be changed manually each step. It is also possible to
write and load a script file which can be used to set values of input variables in certain steps automatically. The script file capabilities are explained in section

The current simulation state is the initial state step 0, the memory component is empty besides the root node, because no declarative knowledge was specified in the procedure. The next screenshot shows the memory content after one step.

The initialisation of the eye position sets the gaze direction towards the root element defined in the data definition file. The visualisation writes the memory path eyes.focus which points at the base node root. The state association can be used for debugging purposes. The memory paths eyes.focus and eyes.state should never be modified but eyes.focus can be a useful to check in a rule what the model focusses at the moment. An additional association "gaze_yaw" is written, which contains a numeric value (measured in degree, positive value: rotated left, negative: rotated right) which corresponds to the rotation of the head compared to the body direction of the model. This can be very useful for modeling gaze behavior: a driver model on a multi-lane highway has to observe the street ahead through the windshield. Looking ahead is very important to recognize possible critical situations. But drivers and also a realistic driver model does not only focus a potential lead car all the time, it also observes cars ahead on the lanes to his right and left. The gaze_yaw attribute allows to specify a rule which fires generally whenever the model looks ahead which means that the gaze direction should be within a certain angular space, e.g. 

For the next step the attributes of the input variable car_1 are set. Section 2.2.4 has already explained that the model uses an ego-centric coordinate system. The coordinate of the root aoi is defined at the location (0,0,-1).

Because the model assumes to be positioned at (0,0,0) it looks into the direction of the negative z axis. In this example we now set the input object at a location where it can be perceived without any necessary movement. Therefore the coordinate (0,0,-2) is set and the id is changed from 0 to 1 which is necessary to signalize that a new object is available at the input channel car_1.

After setting the 4 input values and pressing Enter at each textfield another step is done. The model now has perceived an object at coordinate (0,0,-2) and it can now be seen in the memory. After some

10 The +- 5 degree space is just an example and an adequate value must be chosen by the modeler.
more steps (picture below) the model has recognized the object and adds the default attributes \textit{isa}, \textit{attended}, \textit{visible}, \textit{disappeared} and \textit{ang_size_roc}. The last two attributes will not be explained in this tutorial. \textit{isa} and \textit{attended} have been explained in the section about the visual perception (2.2.1), but there is another attribute called \textit{visible}. This attribute has the value \textit{yes} if it is inside the visual field and \textit{no} if it is not visible at the moment.

The most simple procedure file that can be used to simulate these first steps is the following:

```cascas
version(1.1)

motor{
    eye_start(root)
}

rule(id=1, goal=acquire_task, type=regular){
    -->
}
```

The first extension to the rule set is to detect new \textit{car} objects with a regular rule and shift visual attention towards this object.

```cascas
rule(id=1, goal=acquire_task, type=regular){
    Condition(car.attended == 'no')
    -->
    LookAt(car)
    Memorize(car.attended, 'yes')
}
rule(id=2, goal=acquire_task, type=waiting){
    -->
}
```

Rule 1 checks for cars that were not attended previously and initiates the shift of attention. It also sets the attended attribute to 'yes' which prevents that the rule will fire again for this object. Additionally we
will change the coordinates of the object slightly. The visual perception will calculate the eye movement. To simulate the next steps we switch to the Datapool view.

This view gives an overview about some internal actions of the model, it shows the current goal and the rule that is currently selected. It also shows the focus instrument (vision). The model has now selected rule one and it will take one step until the rule is fired. This can be seen next:

The vision does no longer fixate the root node, the fixation has been released. From this point until the eye movement to the new destination is finished the value -1 is shown (vision), which symbolizes that no objects can be read during the period of movement time. Because the object has moved only a very small distance the eye movement is already finished during the next simulation step (see below).

We will now switch back to the memory view to see what has happened to the internal representation. There are three things that have changed in the picture below:
17. The $x$ and $z$ coordinates have changed to the new values.

18. The `gaze_yaw` value changed and shows the angular difference.

19. The association `focus` now points at the object `car` with the id 1.

If you change the coordinates slightly in the textfields you can now observe that the gaze angle will change. The model has the capability to automatically follow moving objects using (micro) saccades. The next step is to add a second object to the simulation. So far the data definition file contains only one channel to receive cars simultaneously. It is possible to send another car via the already existing channel, but in that case the car that is already visible (car#id=1) can not be transmitted. Therefore we extend the data definition file with a second channel:

```xml
<variable_definition>
    <variable name="car_1" type="Car" mode="in">
        <options> </options>
    </variable>
    <variable name="car_2" type="Car" mode="in">
        <options> </options>
    </variable>
</variable_definition>
```

After the simulation is restarted, input values of the additional channel `car_2` can also be seen and modified.
Initially the values for the input channel \texttt{car\_1} are set before some simulation steps are done. If the model focusses the object (step 8), a second object is put into the second channel. This can be seen in the next figure:

It is important that the \texttt{id} of \texttt{car\_2.id} is different from the already existing object with the \texttt{id 1}. The \texttt{id}'s for new object must be different over the whole simulation, therefore if you have a driving or a flight simulator you need an object counter over all objects over all channels that the model should receive. If the channels can receive different data types, the object ids must be unique over all those objects, e.g. cars, traffic signs, etc.

If the simulation is executed one more step, the object from the second channel is initially recognized and the coordinates plus the id are written into the memory.
The focus is already at car 1 (because rule 1 was fired once). If the simulation is continued some more steps the object type is recognized and the default attributes are added again, which includes the attended = 'no' attribute. It can be imagined what happens: rule 1 will match again and can be selected and fired which will shift the visual attention towards this new object (see picture below). You can now manipulate the coordinates of both objects, they will update, until they move out of the visual field.

The last short section of this introductory documentation describes the script file for the Event Scheduler which can be used to simulate changes on the input channels and to check if certain actions of the model have been done.
3.3.1 Script File

The script file can be used to program small simulation procedures. The language provides simple elements to set input variables in each simulation step. It also offers some commands to check if a certain rule was fired or if a certain memory path contains a value. The script file is automatically loaded by the event scheduler. The script file itself is a simple ASCII text file, which contains a command each line. Each command line starts with the step definition, followed by the command definition cmd. The third parameter is always a variable var (or memory path) that should be checked or modified and finally a value is given:

- **step**: a numeric value >=0 (type integer), which states the step where the command is executed by the Event Scheduler.
- **cmd**: the command that is executed in the given step. The possible commands are:
  - **set**: command to set a value of an attribute of an input channel.
  - **assert**: command to check for a specific value of any attribute of either any output or input channel.
  - **mem_assert**: command to check the value of any memory path.
- **var**: specifies the memory path or input/output channel attribute that should be checked or modified.
- **value**: the value that should be checked or modified for the memory path or input/output channel specified by var.

```
step=5, cmd=set, var=car_1.id, value=1
```

4 Bibliography


Part 1: An Introduction to CASCaS

