BOSCH

VCA Task Script Language

Version 1.8

en Script language

Table of Contents

1 Change log

1.1 Version 1.0

Version 1.0 is integrated in firmware 3.0 together with video analytics plugin version 3.0.

1.2 Upgrade to version 1.1

Version 1.1 is integrated in firmware 3.5 together with video analytics plugin version 3.5:
- added definition of `ColorHistogram`
- added state `SimilarToColor` for comparison of object histogram and user-defined histogram
- added states `HasObjectSize`, `HasAspectRatio`, `HasVelocity`, `HasDirection`, and `HasColor` in order to check presence and absence of object properties

1.3 Upgrade to version 1.2

Version 1.2 is integrated in firmware 4.0 together with video analytics plugin version 4.0:
- added definition of `FlowDetector`
- added state `FlowDetected`
- added states `HasFace` and `HadFace`
- added properties `FaceWidth` and `MaxFaceWidth`
- added `Field` options `ObjectSet` and `SetRelation`

1.4 Upgrade to version 1.3

Version 1.3 is integrated in firmware 5.0 together with video analytics plugin version 5.0:
- added definition of `CrowdDensityEstimator`
- added simple state `EstimatedCrowdDensity`

1.5 Upgrade to version 1.4

Version 1.4 is integrated in firmware 5.5 together with video analytics plugin version 5.5:
- added definition of `Counter`

1.6 Upgrade to version 1.5

Version 1.5 is integrated in firmware 5.6 together with video analytics plugin version 5.6:
- added definition of `Resolution`

1.7 Upgrade to version 1.6

Version 1.6 is integrated in firmware 6.0 together with video analytics plugin version 6.0:
- added definition of `MotionDetector`
- added states `HasClass` and `HadClass`
- marked states `HasFace` and `HadFace` as supported only for firmware versions 4.0 to 5.9
- marked properties `FaceWidth` and `MaxFaceWidth` as supported only for firmware versions 4.0 to 5.9

1.8 Upgrade to version 1.7

Version 1.7 is integrated in firmware 6.3 together with video analytics plugin version 6.3:

- changed naming conventions to emphasize that the task script is not only valid for Intelligent Video Analytics (IVA)
- extended `Field` option `ObjectSet` by `FootPoint`
- added option `TriggerPoint` to `Line` and `Route`
- extended number of allowed `Points` for `Line`
- added property `RelativeObjectSize`
- added non-Boolean states `ObjectsInField` and `ObjectsInState` to definition of `SimpleState`
- extended definition of `ObjectState`
- extended definition of `Counter`
- marked Boolean states `SignalLoss` and `SignalTooNoisy` as no longer supported for firmware versions 6.0 and higher

1.9 Upgrade to version 1.8

Version 1.8 is integratd in firmware 6.6 together with video analytics plugin version 6.6:

- added temporary states to user-defined events

2 Definitions

2.1 Events and states

Whereas states are temporal functions, events occur each time a state changes its value. The script language provides both, events and states, because of the different operations defined for them. On the one hand, events allow expressing temporal relations between objects. On the other hand, non-temporal (e.g. spatial) relations of objects are more easily described by states.

2.2 Properties

In order to distinguish between Boolean and non-Boolean states, the notion property is introduced referring to non-Boolean states.

2.3 Rules

When speaking about rules, both events and states are meant which are output by the Alarm Task engine, i.e. which have been declared external. A rule is said to be active if the corresponding state is active or the corresponding event has been triggered. In case of an event, the corresponding rule is active starting with the frame on which the event has been triggered to the next processed frame where the rule is inactive again.

2.4 Alarm Task engine

The Alarm Task engine parses the output of the VCA (Video Content Analysis) module and searches within this stream for rules defined according to the VCA Task script language.

3 System integration

3.1 System overview

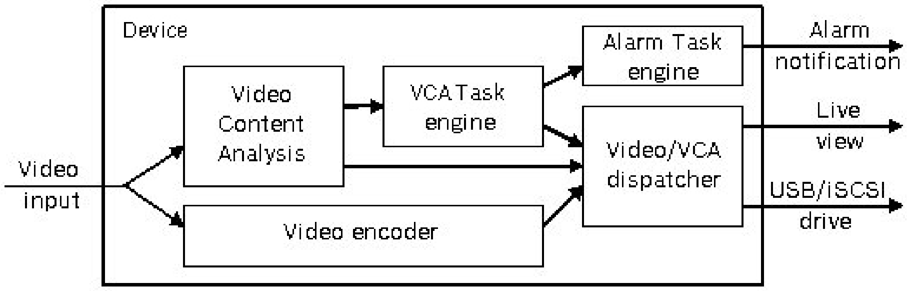

Figure 3.1 Device architecture

The figure "Device architecture" shows the parts of the architecture of the encoding device which are relevant for Video Content Analysis (VCA). First, the video signal is sent to the video encoder and the VCA software. The former outputs e.g. H.263 or H.264 streams, the latter produces a meta stream encoded in the Bosch VCD format. Both outputs are received by the video/VCA dispatcher which forwards them to clients which are interested in the data. Thereby, a recording task is treated as just another client.
The VCA module detects objects in the scene independently of any user-defined rules. The extracted objects and their properties are forwarded as a VCD stream to the Alarm Task engine which does the detection of user-defined rules. The output of the Alarm Task engine are up to 16 alarm flags. On the one hand, these are merged into the VCD stream. On the other hand, the VCA Task engine is notified about changes of alarm flags. The VCA Task engine allows association of any kind of actions to alarm flags; actions like sending e-mail, triggering a relay output, or notification of a video management system.

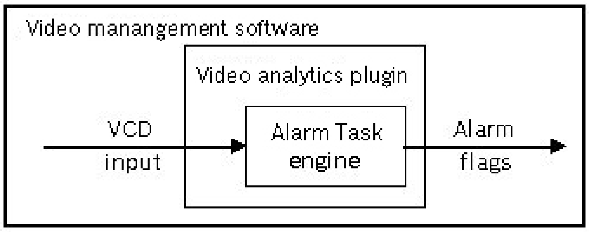

Figure 3.2 Video manangement software architecture

The Alarm Task engine is configured with a script specified in the VCA Task script language. The same language is used in the forensic search applications by Bosch (see figure "Video management software architecture") to look for unforeseen events in recordings. Applications include Bosch Video Management Software, Bosch Video Client or Configuration Manager, for example. There, the recorded VCD streams are fed through another instance of the Alarm Task engine which can run with another set of rules. The occurrences of events are displayed in a timeline which allows fast navigation to relevant video sequences. In order to provide different kinds of forensic search tools, the Alarm Task engine is packed into a plugin which can be easily exchanged. Version 3.0 of the video analytics plugin was the first one supporting the VCA Task script language and configuration of the corresponding Alarm Task engine.

3.2 User-defined tasks

Via the video analytics plugin the Alarm Task engine can be configured. To simplify matters, the plugin provides a comprehensive list of predefined queries with corresponding wizards. The wizards automatically enter VCA Task script code into the Alarm Task engine script. Such code fragments are framed by comments which allow editing of tasks at a later time. Modification of automatically generated code can lead to corrupt tasks which can no longer be interpreted by wizards.
The video anlaytics plugin enables the definition of up to 8 tasks corresponding to the first 8 external rules of the script. However, in order to make a user-defined rule visible in this task list, it must be wrapped as follows:

```
//@Task T:0 V:0 I:<n> “user defined task” {
external ObjectState #<n> := true;
//@}
```

Thereby, `<n>` is a placeholder for the ID of the user-defined rule. Furthermore, there must be no other task with the same ID. The name of the task can be specified within the quotation marks.
In total, up to 16 external rules can be defined by the user with the VCA Task script, but only the first 8 (IDs `1` to `8`) will be visible in the task list. Nevertheless, the other tasks (IDs 9 to `16`) can be used to trigger alarms and hence can be searched for by their name using the Alarm Message search, for example.
Note that the ID defines whether the task is displayed in the task list or not. If for example task `5` is deleted in a script that had tasks `1` to `9` defined, the 5th position will be blank in the list although in total there are still 8 tasks defined.

4 Syntax

4.1 Types

4.1.1 Basic types

The VCA Task script language supports `BOOLEAN`, `INTEGER`, `FLOAT`, `ANGLE` and `OID` as basic types. The domains of these types are as follows: `BOOLEAN`s take either the value `true` or `false`, `INTEGER`s are signed 32-bit integral numbers, `FLOAT`s represent 32-bit floating numbers, `ANGLE`s are specified in degrees, and `OID`s are unique identifiers of objects.
In order to account for the periodic structure of angles, the special type `ANGLE` is introduced. The only operation allowed for angles is a range check. It returns `true` if there exists an equivalent angle within the specified range. Two angles are equivalent if their difference is a multiple of 360 degrees.
There exists a special value `NAN` (not a number) which indicates that the value does not exist. The return value is `NAN` when, for instance, the direction of an object is accessed which does no longer exist.

Example: speeding

The subsequent VCA Task script example checks for all objects if they are moving faster than 30 meters per second; objects which exceed this limit trigger an alarm:

```
external ObjectState #1 := Velocity within(30.0,*);
```

The `Velocity` function returns the speed of an object. The `within` operation takes the corresponding `FLOAT` value and compares it with the left bounded interval `(30.0,*)`. The resulting `BOOLEAN` is assigned to the `ObjectState #1` of the considered object.

4.1.2 Attributes

Events and states can be augmented with attributes. Each attribute is characterized by a basic type and a unique name. The name is needed in order to distinguish attributes of the same type and to access the attribute. When introducing events, the following syntax will be used to describe their attributes:

```
<event> -> ( <type> : <name>, ... , <type> : <name> )
```

For states, a similar syntax is used:

```
<state>( <type> : <name>, ... , <type> : <name> ) -> <type>
```

The attribute list can be empty in both cases.

Example: crossing two lines

The subsequent VCA Task script defines two lines and triggers `Event #1` if the same object passes the first line and afterwards the second line. In order to check whether both `CrossedLine` events have been triggered by the same object, the `oid` attributes of both events are compared. Without this condition, `Event #1` would be triggered even if 1 object had passed `Line #1` before a completely different crossed `Line #2`. The keywords `first` and `second` allow access to the attribute lists of the two events involved in a `before` relation.

```
Line #1 := {
   Point(10,10) Point(10,50) Direction(0)
};
Line #2 := {
   Point(50,10) Point(50,50) Direction(0)
};
external Event #1 := {
      CrossedLine #1 before CrossedLine #2
      where first.oid == second.oid
};
```

4.2 Comments

The VCA Task script language supports C-style comments. That means, all characters of a line following `//` (two slashes) are ignored by the Alarm Task engine as well as all characters which are framed by `/*` (slash and asterisk) and `*/` (asterisk and slash). The latter kind of commenting can be used to comment out several lines at once or a part of a single line.

Example

```
Field #1 := { // this is a comment
   Point(10,10) /*Point(10,50)*/ Point(50,50)
   Point(50,10)
};
/* all characters
   within this block
   are commented out */
```

4.3 Script normalization

The VCA Task script language provides a method to normalize coordinates within a specified range. Place the keyword `Resolution` at the beginning of the script. The `Min` coordinate defines the upper left corner of the image and the `Max` coordinate defines the lower right corner. All following coordinates are then defined in relation to `Min` and `Max`.

Example

```
Resolution := { Min{0,0} Max{1,1} };
Line #1 := {
   Point(0.10,0.10) Point(0.10,0.50)
   Point(0.50,0.50) Point(0.50,0.10)
};
```

4.4 Primitives

The VCA Task script language supports several geometrical primitives. These primitives observe single objects and trigger events on certain actions. The number of primitives which can be instantiated is limited by the memory of the device on which the Alarm Task engine is running. In the firmware, this limits the number of routes to 8, and the total number of primitives to 32.

4.4.1 Field

Syntax

`Field` primitives are defined in the following way:

```
Field #<n> := {
   Point(<x>,<y>) ... Point(<x>,<y>)
   [DebounceTime(<time>)]
   [ObjectSet(<objectset>)]
   [SetRelation(<relation>)]
};
```

`<n>` is the number of the field and must be between `1` and `32`. The specified points span a polygon. This polygon must have between 3 and 16 points and must be simple, i.e. it must not intersect with itself. The coordinates of the points are specified in pixels with the image resolution processed by the VCA algorithm. `<time>` specifies the optional `DebounceTime` in seconds. Its default value is `0`. When the `DebounceTime` is set, the state of an object, whether it is inside or outside of the field, only changes if the object stays on the other side for at least the time specified by `<time>`. This way, one can get rid of positional errors in the object tracking or multiple alarms of objects moving along the border of a field.
If a flow detector is used, the `DebounceTime` has a different meaning. The `<time>` specifies the post-alarm time for the field. This ensures that if many short flow alarms are detected within the debounce time `<time>` they are merged to a long alarm period.
`ObjectSet` and `SetRelation` specify which parts of the object are considered to determine whether it is regarded as inside or outside. Thereby, `<objectset>` can be set to either `BaryCenter`, `FootPoint` or `BoundingBox` with `BaryCenter` as default value. The `<relation>` can be switched between `Intersection` and `Covering` with `Intersection` as default value. For instance, if `BoundingBox` and `Covering` are selected, an object's bounding box must be completely inside the specified polygon to be regarded as inside. If `<objectset>` is set to `BaryCenter`, both `<relation>` options result in the same behavior since the object set consists of a single point only.

Trigger overview:

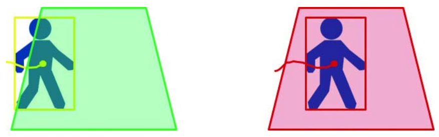

Figure 4.1 `ObjectSet (BoundingBox)` with `SetRelation (Covering)`

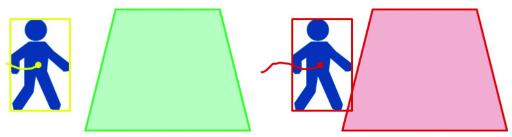

Figure 4.2 `ObjectSet (BoundingBox)` with `SetRelation (Intersection)`

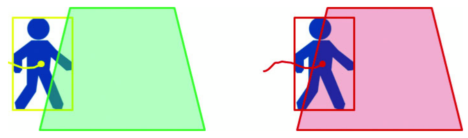

Figure 4.3 `ObjectSet (BaryCenter)` with `SetRelation (Intersection)` or `(Covering)`

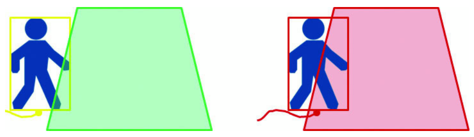

Figure 4.4 `ObjectSet (FootPoint)` with `SetRelation (Intersection)` or `(Covering)`

States

```
InsideField #<n>(OID:oid) -> BOOLEAN
```

The state `InsideField #<n>` is set if object `oid` is inside `Field #<n>`.

```
ObjectsInField #<n> -> INTEGER
```

`ObjectsInField #<n>` returns the number of objects which are currently in `Field #<n>`.

Events

```
EnteredField #<n> -> (OID:oid)
```

The event `EnteredField` is triggered when an object enters `Field #<n>`. The object which has caused the event can be queried by the argument `oid`.

```
LeftField #<n> -> (OID:oid)
```

The event `LeftField` is triggered when an object leaves `Field #<n>`. The object which has caused the event can be queried by the attribute `oid`.
Note that an object does not trigger the corresponding `EnteredField #<n>` event when it is already within a field at first detection.

Example
The following example triggers an alarm if the same object has first entered the specified field and later left it again.

```
Field #1 := {
   Point(10,10) Point(10,50)
   Point(50,50) Point(50,10)
};
external Event #1 := {
   EnteredField #1 before LeftField #1
   where first.oid == second.oid
};
```

4.4.2 Line

Syntax

`Line` primitives are defined in the following way:

```
Line #<n> := {
   Point(<x>,<y>) Point(<x>,<y>)
   Direction(<dir>)
   [DebounceTime(<time>)]
   [TriggerPoint (<triggerpoint>)]
};
```

`<n>` is the number of the line and must be between `1` and `16`. A line has at least 2 and at most 16 points whose coordinates are specified in pixels with the image resolution processed by the VCA algorithm. With the argument `<dir>`, one can choose whether any object which passes the line triggers an event or whether only objects which pass from left to right respectively right to left are relevant. In the first case, `<dir>` is expected to be `0`. In the latter cases, `<dir>` takes the value `1` respectively `2`. `<time>` specifies the optional `DebounceTime` in seconds. Its default value is `0`. When the `DebounceTime` is set, a `CrossedLine #<n>` event is only triggered if the same object will not cross the same line in the opposite direction within the specified time window `<time>` afterwards. This way, one can get rid of positional errors in the object tracking or multiple alarms of objects moving along the line. The `TriggerPoint` specifies which point of the object is considered to trigger a line crossing. Possible trigger points are `BaryCenter` or `FootPoint`.

Trigger overview:

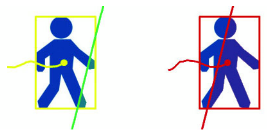

Figure 4.5 `TriggerPoint (BaryCenter)`

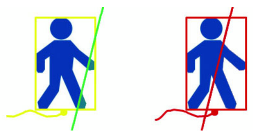

Figure 4.6 `ObjectSet (FootPoint)`

Events

```
CrossedLine #<n> -> (OID:oid)
```

The event `CrossedLine #<n>` is triggered when an object crosses `Line #<n>` in the specified way. The object which has caused the event can be queried by the attribute `oid`.

Example

See *Section Example: crossing two lines, page 11.*

4.4.3 Route

Syntax

Route primitives are defined in the following way:

```
Route #<n> := {
   Point(<x>,<y>) Distance(<r>)
   ...
   Point(<x>,<y>) Distance(<r>)
   Direction(<dir>)
   MinPercentage(<min>)
   MaxGap(<max>)
   [TriggerPoint (<triggerpoint>)]
};
```

<n> is the number of the route and must be between 1 and 32. A route has at least two and at most 8 points. The coordinates of the points are specified in pixels with the image resolution processed by the VCA algorithm. Each point is followed by a tolerance <r>. With these tolerances, the path of points is broadened to a stripe. Objects which move along the stripe in the specified direction trigger the event FollowedRoute #<n>. If <dir> is set to 1, only object movements are considered which go from the first towards the last point. Is <dir> set to 2, only object movements are considered which go in the opposite direction. If <dir> equals 0, any object movement within the stripe is taken into account. Object movements which do not satisfy the directional constraint are ignored.
The parameters MinPercentage and MaxGap specify the tolerance of the detector. If Direction is set to 0, the meaning of the two parameters is as follows. The detector remembers for each object which parts of the stripe have been visited. If more than MinPercentage of the whole stripe are visited and the largest gap between two visited parts (including the gaps at the very beginning and the very end of the stripe) is smaller than MaxGap, then the FollowedRoute #<n> event is triggered. If a direction has been assigned to the route, object movements within the route are only taken into account if the movement fits the specified direction and if the distance to the last visited part is not larger than MaxGap.
The TriggerPoint specifies which point of the object needs to follow the route. Possible trigger points are BaryCenter or FootPoint.

Trigger overview:

Figure 4.7 On the left: TriggerPoint (BaryCenter); on the right: TriggerPoint (FootPoint)

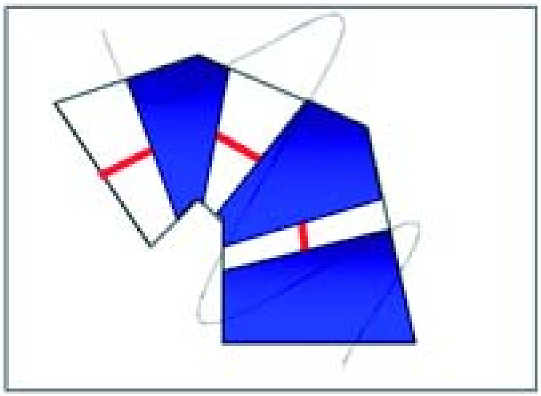

Figure 4.8 Illustration of an object following a predefined route

The figure above illustrates an object following a predefined route. The parts of the route which have been visited by the object are marked in blue. The gaps in between are highlighted with a red line.

Events

```
FollowedRoute #<n> -> (OID:oid)
```

The event `FollowedRoute #<n>` is triggered when an object has followed the `Route #<n>`. The object which has caused the event can be queried by the attribute `oid`.

Example

```
Route #1 := {
   Point(20,20) Distance(5)
   Point(20,50) Distance(5)
   Point(50,50) Distance(5)
   Direction(0)
   MinPercentage(90)
   MaxGap(20)
};
external Event #1 := FollowedRoute #1;
```

4.4.4 Loitering

Syntax

`Loitering` primitives are defined in the following way:

```
Loitering #<n> := {
   Radius(<r>)
   Time(<time>)
};
```

`<n>` is the number of the `Loitering` primitive and must be between `1` and `32`. The `Loitering` primitive detects objects which stay at one place for `<time>` seconds. `<r>` specifies the spatial tolerance in meters of the loitering detector. For the measurement of object movements in meters, the camera must have been calibrated beforehand.

States

```
 IsLoitering #<n> (OID:oid) -> BOOLEAN
```

The state `IsLoitering #<n>` of an object `oid` is set when the object stays at the same place for at least the specified time.

Example

```
Loitering #1 := {
   Radius(5)
   Time(10)
};
external ObjectState #1 := IsLoitering #1;
```

4.4.5 ColorHistogram

Syntax

`ColorHistogram` primitives are defined in the following way:

```
ColorHistogram #<n> := {
   HSV(<h>,<s>,<v>[,<weight>])
   ... HSV(<h>,<s>,<v>[,<weight>])
   Similarity(<similarity>)
   Outlier(<outlier>)
};
```

`<n>` is the number of the `ColorHistogram` primitive and must be between `1` and `32`. The `ColorHistogram` primitive compares object colors with user-defined colors. Up to 5 basic colors can be selected with the `HSV` keyword. The default value of the optional parameter `<weight>` is `1`. The total weight of the basic colors must not exceed `255`. Basic colors are defined in the HSV color space with `<h>` (a value between `0` and `360`) representing the hue component, `<s>` (a value between `0` and `100`) the saturation, and `<v>` (a value between `0` and `100`) the intensity. The figure "HSV cone" visualizes the 3 components of the HSV color space.

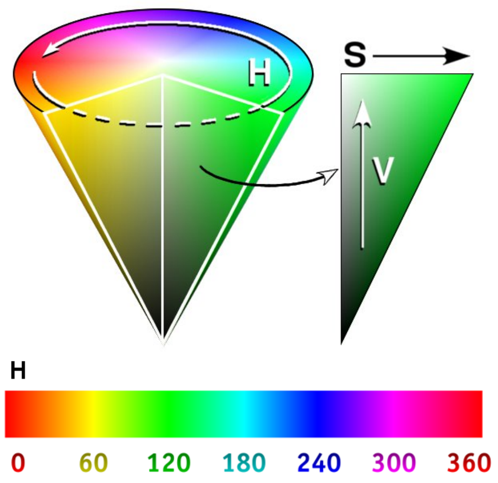

Figure 4.9 HSV cone

`<similarity>` is a value between `0` and `100` and specifies how similar a color histogram must be in order to be regarded as a match. The more similar two histograms are the larger is their `<similarity>`. The parameter `<outlier>` allows partial matches between the user-defined colors and the object’s color histogram, i.e. the user-defined colors cover only a subset of the object’s color histogram and the remaining colors should be regarded as outliers. For instance, when looking for persons wearing a red jacket, about 50% of the object’s colors should be red and the other 50% are not taken into account.

States

```
 SimilarToColor #<n> (OID:oid) -> BOOLEAN
```

The state `SimilarToColor #<n>` of an object `oid` is set when the latest color histogram of this object is at least as similar to the user-defined color histogram as the specified threshold.

Example
The following color histogram detects objects which contain at least 25% reddish colors and at least 25% dark colors.

```
ColorHistogram #1 := {
   HSV(0,100,100)
   HSV(0,0,0)
   Similarity(90)
   Outliers(50)
};
external ObjectState #1 := SimilarToColor #1;
```

4.4.6 FlowDetector

Syntax
`FlowDetector` primitives can be defined in one of the following ways:

```
FlowDetector #<n> := {
   [Direction(<minangle>,<maxangle>)]
   [Direction(<minangle>,<maxangle>)]
   [Velocity(<minvelocity>,<maxvelocity>)]
   [Activity(<minactivity>,<maxactivity>)]
   [Field #<m>]
};
```

or

```
FlowDetector #<n> := {
   CounterFlow(<timewindow>,<angletol>)
   [Velocity(<minvelocity>,<maxvelocity>)]
   [Activity(<minactivity>,<maxactivity>)]
   [Field #<m>]
};
```

The flow detector operates on the significant flow field computed by the VCA algorithms. Each detected flow vector has to pass a set of user-defined filters before it triggers an alarm. In the first definition, up to two directional filters can be defined. If 2 directions are specified, a motion vector must pass at least 1 for further processing. Each directional filter consists of a `<minangle>` and `<maxangle>` specified in degrees. The coordinate system for angles is shown in the figure "Image coordinates" in chapter Object properties.

In the second definition, the direction is automatically determined based on a main flow analysis. The last `<timewindow>` seconds are considered in order to compute the main flow direction. As soon as a dominant direction is detected, this direction defines the main flow and activates the flow detector. If there is no dominant direction within the last `<timewindow>` seconds, the flow detector is inactive. A motion vector going in the opposite direction to the main flow with an angular tolerance of at most `<angletol>` degrees passes the directional filter of the second definition. If a spatial constraint is specified, only motion vectors within `Field #<m>` are taken into account when estimating the main flow direction. The other filters do not apply to the main flow estimation.
Additional filters for the velocity, activity, and space can be set up. Thereby, the `<minvelocity>` and `<maxvelocity>` are specified in pixels per second. If a `Field #<m>` is added to the definition, motion vectors outside the `Field #<m>` are ignored during processing. If no field is added, all motion vectors are considered. The activity measures the number of active motion vectors, i.e. the number of motion vectors which have passed all filters. Thereby, the activity is `0` if no motion vector is active. The maximum activity of `100` is reached when the complete specified field is filled with active motion vectors. With `<minactivity>` and `<maxactivity>` the user can define an activation interval within which the flow detector triggers an alarm. Thereby, the lower bound `<minactivity>` is excluded from the interval. This implies that the flow detector will only trigger an alarm if there is any motion.
If a `Field #<m>` is present in the definition of a flow detector, the flow detector inherits the `DebounceTime` of the specified field. Thereby, the meaning of the `DebounceTime` is that the flow field must pass all the constraints for at least `DebounceTime` many seconds before the flow detector will trigger an alarm.

States

```
DetectedFlow #<n> -> BOOLEAN
```

The state `DetectedFlow #<n>` is set when there is a significant flow fulfilling all the constraints defined in the corresponding `FlowDetector #<n>` definition.

Example
The following example triggers an alarm if a significant motion from right to left has been detected within the specified rectangle.

```
Field #1 := {
   Point(10,10) Point(10,50)
   Point(50,50) Point(50,10)
};
FlowDetector #1 := {
   Direction(-45,45)
   Field #1
};
external SimpleState #1 := DetectedFlow #1;
```

4.4.7 CrowdDensityEstimator

Syntax

`CrowdDensityEstimator` primitives are defined in the following way:

```
CrowdDensityEstimator #<n> := {
   [Activity(<minactivity>,<maxactivity>)]
   CrowdDensityField #n
   [DebounceTime(<seconds>)|SmoothingTime(<seconds>)]
};
```

The crowd level estimation detector uses the reference image which should show the empty scene to detect the crowd in front of the background reference. This scene is limited by the `CrowdDensityField` which must be specified in advance. The VCA algorithm calculates a crowd level activity value for the given region which goes from zero to 100%. The trigger activity level can be limited by setting the activation interval `<minactivity>` and `<maxactivity>`. The crowd activity level is calculated every second. To ignore a short-term crowd level jump, the `DebounceTime` or `SmoothingTime` filter can be set. If `DebounceTime` is used, an alarm is triggered if the crowd level is within the activity thresholds for more than the time specified. With `SmoothingTime`, `<seconds>` defines a sliding window: an alarm is triggered if the average over the time specified is within the activity thresholds.

States

```
EstimatedCrowdDensity #<n> -> BOOLEAN
```

The state `EstimatedCrowdDensity #<n>` is set when there is a significant crowd level fulfilling the constraints defined in the corresponding `CrowdDensityEstimator #<n>` definition.

Example

The following example triggers an alarm if the crowd level is greater or equal to 25% for more than 10 seconds within the crowd field that has to be specified and saved to the device before the query or the recording is done.

```
CrowdDensityEstimator #1 := {
   Activity(25,100)
   CrowdDensityField #1
   DebounceTime(10)
};
external SimpleState #1 := EstimatedCrowdDensity #1;
```

4.4.8 MotionDetector

Syntax

`MotionDetector` primitives are defined in the following way:

```
MotionDetector #<n> := {
   [Activity(<minactivity>,<maxactivity>)]
   [Size(<minsize>,<maxsize>)]
   [Field #n]
};
```

The motion detector triggers an alarm if the provided motion cells fulfill the defined criteria.The `MotionDetector` can operate on the whole screen or on the specified `Field`. The trigger activity level can be limited by setting the activation interval `<minactivity>` and `<maxactivity>`. The `Activity` is specified in percent of the selected area. Furthermore, the trigger activity level can by limited by the `Size` of the cell cluster. This interval `(<minsize>,<maxsize>)` is specified in percent of the whole screen.

States

```
 DetectedMotion #<n> -> BOOLEAN
```

The state `DetectedMotion #<n>` is set when there is a significant motion level fulfilling the constraints defined in the corresponding `MotionDetector #<n>` definition.

Example

The following example triggers an alarm if the size of the cell cluster is greater or equal to 0.5% of the whole screen.

```
MotionDetector #1 := {
   Size(0.5,100)
};
external SimpleState #1 := DetectedMotion #1;
```

4.5 Object-specific events

The main task of the VCA algorithm is object detection and object tracking. Besides the position and other properties of the currently detected objects, the algorithm notifies about basic object events. When the algorithm detects a new object, an `Appeared` event is triggered. Correspondingly, a `Disappeared` event is sent as soon as an object gets lost. `Idled` and `Removed` are special cases of disappearing objects with the following meaning. If an object does not move at all for a certain time, an `Idled` event is triggered. This happens if a person leaves an object like a bag. Similarly, an object can be picked up by a person triggering a `Removed` event.

4.5.1 Object events

Object events are triggered by appeared or disappeared objects.

Events

```
Appeared (OID:oid)
Disappeared (OID:oid)
Idled (OID:oid)
Removed (OID:oid)
```

4.6 Counter

Syntax

A counter is defined in one of the following ways:

```
Counter #<n> := {
   Event #<m>
};
```

or

```
Counter #<n> := {
   Counter #<x> + Counter #<y>
};
```

or

```
Counter #<n> := {
   Counter #<x> - Counter #<y>
};
```

or

```
Counter #<n> := {
   <num-expr>
};
```

or

```
external Counter #<n> := {
...
};
```

The counter can either count the number of triggered events or a counter can be assigned the sum or difference of two counters or a non-Boolean state. Moreover, the keyword `external` indicates that the counter value is added to the VCD stream as well as included in the RCP counter message. Hence, without this keyword the counters are only used internally and the current value cannot be displayed. Up to 32 counters can be configured. The default range of each counter is between `0` and the maximum of a 32-bit value.
The following optional arguments can be set: Each counter can be named via `Text("<string>")` in which the string is a 32-byte UTF-8 encoded string. The counter allows two different kinds of modes – `Mode(KeepMax)` or `Mode(Wraparound)`. If `KeepMax` mode is set, the counter stops at the upper bound. In `Wraparound` mode the counter restarts from the lower bound (e.g. `0`). The minimum and maximum bound can be set with the argument `within(<min>,<max>)` in which `<min>` corresponds to the lower bound and `<max>` to the upper bound. The position of the counter value displayed in the video can be defined by the keyword `TopLeft(<x>,<y>)` which indicates the position within the frame. The origin is in the upper left corner.
Generally, a counter does not trigger an alarm. In order to do so, the counter needs to be assigned to a state as shown below:

```
external SimpleState #<n> := Counter #<n> within(<min>,<max>)
```

Therefore, an alarm is triggered as soon as the counter value is within the boundary defined by `<min>` and `<max>`.

Example

```
ObjectState #32 := true;
Event #2 := OnSet ObjectState #32;
external Counter #5 := {
   Event #2 Text("Every 5th:")
   TopLeft(4,14) within(0,5) Mode(Wraparound)
};
external SimpleState #2 := Counter #5 within(5,*);
```

4.7 Object properties

Each tracked object has a set of properties. These properties include the object’s position, its direction of movement, its speed, its size, and its bounding box. Whereas the position is only indirectly accessible via geometrical primitives, the other properties are directly available in conditional expressions via the subsequent functions. All the following functions return `NAN` if either the object does not have this property or if the object does not exist at all. For each property there exists a below mentioned state which returns `true` if the property is present and `false` otherwise. For all subsequent functions and states, the `oid` is optional if the attribute list of the current scope contains the argument `OID:oid`.

Functions

```
 Direction (OID:oid) -> ANGLE
```

This function returns the current direction in degrees of object `oid`. If no direction is available for this object, the result is `NAN`. Directions are expressed in image coordinates. Thereby, a movement from the right image border to the left border corresponds to zero degrees, from the top border to the bottom border to 90 degrees, from the left border to the right border to

180 degrees, and from the bottom border to the top border to 270 degrees (see figure "Image coordinates").

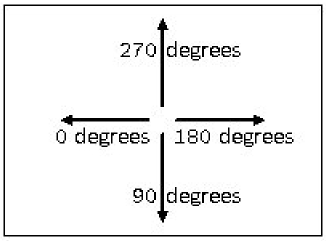

Figure 4.10 Image coordinates

```
Velocity (OID:oid) -> FLOAT
```

This function returns the current velocity in meters per second of object `oid`. The speed is estimated using the object's translation in object coordinates and camera calibration parameters. If no velocity is available for this object, the result is `NAN`.

```
AspectRatio (OID:oid) -> FLOAT
```

This function returns the current aspect ratio of object `oid`. It is defined as the ratio of height and width of the object's bounding box. If no bounding box is available for this object, the result is `NAN`. A square has an aspect ratio of `1`. An object which is two times higher than wide, has an aspect ratio of `2` (see figure "Current aspect ratio").

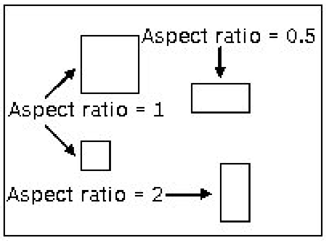

Figure 4.11 Current aspect ratio

```
ObjectSize (OID:oid) -> FLOAT
```

This function returns the current size in square meters of object `oid`. The size is estimated based on the shape of the object and camera calibration parameters. If no size is available for this object, the result is `NAN`.

```
RelativeObjectSize (OID:oid) -> FLOAT
```

This function returns the current size relative to the screen size (from 0.000 to 1.000) of object `oid`. The size is calculated based on the shape of the object.

The following functions are only supported for firmware versions 4.0 to 5.9:

```
FaceWidth (OID:oid) -> FLOAT
```

This function returns the current width of a detected head in pixels assigned to object `oid`. If no face has been detected for this object, the result is `NAN`.

```
MaxFaceWidth (OID:oid) -> FLOAT
```

This function returns the maximum width over all so far detected heads assigned to object `oid`. If there was no detection for this object so far, the result is `NAN`.

4.7.1 States

```
HasDirection (OID:oid) -> BOOLEAN
```

returns `true` if the direction of the object with object ID is available for the current frame. If the object is not present in the current frame, the return value is `false`.

```
HasVelocity (OID:oid) -> BOOLEAN
```

returns `true` if the velocity of the object with object ID is available for the current frame. If the object is not present in the current frame, the return value is `false`.

```
HasAspectRatio (OID:oid) -> BOOLEAN
```

returns `true` if a bounding box of the object with object ID is available for the current frame. If the object is not present in the current frame, the return value is `false`.

```
HasObjectSize (OID:oid) -> BOOLEAN
```

returns `true` if a bounding box of the object with object ID is available for the current frame. If the object is not present in the current frame, the return value is `false`.

```
HasColor (OID:oid) -> BOOLEAN
```

returns `true` if a color histogram of the object with object ID is available since the last appearance of the object. If the object is not present in the current frame, the return value is `false`.

```
HasClass (<classnames>) -> BOOLEAN
```

returns `true` if an object matching one of the specified classes is detected in the current frame. If no such object is present in the current frame, the return value is `false`.

```
HadClass (<classnames>) -> BOOLEAN
```

returns `true` if an object matching one of the specified classes has been detected since the last appearance of the object. If no such object is present in the current frame, the return value is `false`.
For the last two states, classes are defined entering the required `<classnames>` from the following list separated by empty space:
- `Person` (moving upright persons)
- `Bike` (moving bycicles)
- `Car` (moving cars)
- `Truck` (moving trucks/big cars)
- `NoClass` (non of the above classes)

The following functions are only supported for firmware versions 4.0 to 5.9:

```
HasFace (OID:oid) -> BOOLEAN
```

returns `true` if a head is detected on the object with object ID in the current frame. If the object is not present in the current frame, the return value is `false`.

```
HadFace (OID:oid) -> BOOLEAN
```

returns `true` if a head has been detected on the object with object ID since the last appearance of the object. If the object is not present in the current frame, the return value is `false`.

4.8 Tamper states

The VCA algorithms can detect tampering of cameras. The output of the tamper detection is available via the following Boolean states:

```
SignalTooDark -> BOOLEAN
```

If the video signal becomes too dark, such that automatic object detection becomes impossible, the `SignalTooDark` state is enabled. This can happen if the camera is covered by a sheet, such that almost black images are recorded.

```
SignalTooBright -> BOOLEAN
```

If the video signal becomes too bright, such that automatic object detection becomes impossible, the `SignalTooBright` state is enabled. This can happen if the camera is dazzled by a strong light source.

```
GlobalChange -> BOOLEAN
```

If most of the image content has changed, the `GlobalChange` state is enabled. This can happen if the camera is moved or if an object comes too close to the camera.

```
RefImageCheckFailed -> BOOLEAN
```

If a reference image has been set during the configuration of the VCA algorithm and the reference checking of the VCA algorithm has been enabled, the algorithm detects manipulations of the camera by comparing the current video signal with the preset reference image. Significant differences between the two images are recorded in the VCD stream. This information can be accessed from the VCA Task script language via the `RefImageCheckFailed` state.

The following two Boolean states are no longer supported for firmware versions 6.3 and higher:

```
SignalLoss -> BOOLEAN
```

If the video signal of an encoder device is lost, the `SignalLoss` state is set.

```
SignalTooNoisy -> BOOLEAN
```

If the video signal becomes too noisy, such that automatic object detection becomes impossible, the `SignalTooNoisy` state is enabled. This can happen if an analog video signal is transmitted over a large distance or if the sensitivity of the camera sensor is not sufficient in night vision applications.

Example

For firmware 6.3, the following VCA Task script triggers an alarm on any detected tamper action.

```
external SimpleState #1 :=
  SignalTooDark or SignalTooBright or
  GlobalChange or RefImageCheckFailed;
```

Note that in this example it is important to work with a user-defined `SimpleState` instead of an `ObjectState`, since the latter is only evaluated for detected objects. However, when the camera is tampered there are usually no detections.

4.9 User-defined states

User-defined states are always Boolean states, i.e. these states take either the value `true` or `false`. A user-defined state has either no attributes or the object ID as an attribute. The former state is called `SimpleState`, whereas the latter is called `ObjectState`. In the subsequent sections their syntax and usage is explained in more detail.

4.9.1 SimpleState

A `SimpleState` is defined in the following way:

```
[external] SimpleState #<n> := <condition>;
```

Thereby, `<condition>` is a placeholder for a Boolean expression. The user can specify up to 32 states (from `1` to `32`) via the number `<n>`, but only the first 16 states (`<n>` from `1` to `16`) can be `external`. The same `SimpleState` cannot be defined twice. The attribute list of a `SimpleState` is always empty. Therefore, one cannot access object specific properties in the condition clause of a `SimpleState`.
The following table summarizes the predefined states which can be used instead of the `<condition>` placeholder besides a previously defined `SimpleState`:

```
SignalTooDark
SignalTooBright
GlobalChange
RefImageCheckFailed
```

The following predefined states are not supported for firmware versions 6.3 and higher:

`SignalLoss`	Note: only for encoder devices
`SignalTooNoisy`	

4.9.2 Boolean composition of conditions

Several conditions can be composed. The syntax of conjunctions is as follows:

```
<condition> := <condition> and <condition>
<condition> := <condition> && <condition>
```

Similarly, disjunctions are written as:

```
<condition> := <condition> or <condition>
<condition> := <condition> || <condition>
```

The negation of conditions is:

```
<condition> := not <condition>
<condition> := !<condition>
```

Ambiguities in the evaluation of expressions are resolved by priorities. Negations have the highest priority, followed by conjunctions, and disjunctions last. Furthermore, the priority can be controlled by embracing sub-expressions with brackets:

```
<condition> := ( <condition> )
```

4.9.3 Properties in conditions

Properties and non-Boolean states can be used as condition in the following way:

```
<condition> := <num-expr> within(<min>,<max>)
```

The `within` keyword checks if the specified `<num-expr>` is within the specified interval. The interval bounds, `<min>` and `<max>`, are constant values. If 1 of the 2 bounds is replaced by an asterisk `*`, the corresponding bound is ignored. A `<num-expr>` can either be an attribute, a constant value, a non-Boolean state or a property. In combination with the `within` keyword, `<num-expr>` must be either of type `INTEGER`, `FLOAT`, or `ANGLE`.
Besides the `within` relation, a simple equation is allowed for any non-Boolean type as long as both operands are of the same type:

```
<condition> := <num-expr> == <num-expr>
<condition> := <num-expr> != <num-expr>
```

`ObjectsOnScreen`, `ObjectsInField`, and `ObjectsInState` are non-Boolean states which can be used in the definition of a `SimpleState #<n>`. It returns the number of objects which are currently detected by the VCA algorithm on the screen respectively in a field, or which trigger an object state.
Note that the conditions described in this section return `false` if the value of 1 `<num-expr>` is `NAN`.

Example
The following VCA Task script triggers an alarm if at least two objects are detected by the VCA algorithm.

```
external SimpleState #1 := ObjectsOnScreen within(2,*);
```

The subsequent example triggers an event if two different objects cross the same line. Thereby, the second object must cross the line within 10 seconds after the first.

```
Line #1 := {
  Point(10,10) Point(50,50)
  Direction(0)
};
external Event #1 := {
  CrossedLine #1 before(0,10) CrossedLine #1
  where first.oid != second.oid
};
```

4.9.4 ObjectState

An `ObjectState` is defined in the following way:

```
[external] ObjectState #<n> := <condition>;
```

Thereby, `<condition>` is a placeholder for a Boolean expression. The user can specify up to 32 states (from `1` to `32`) via the number `<n>`, but only the first 16 states (`<n>` from `1` to `16`) can be `external`. The same `ObjectState` cannot be defined twice.
In contrast to a `SimpleState` which is instantiated only once, an `ObjectState` is instantiated for each object visible in the processed frame. In order to distinguish all the instances, the object ID is associated with each instance as an attribute. Hence, an `ObjectState`'s attribute list is:

```
ObjectState #<n>(OID:oid) -> BOOLEAN
```

In the `<condition>` clause, this object ID can be used to access object properties or an earlier defined `ObjectState` of the same object in addition to the ones which are allowed within the `<condition>` clause of a `SimpleState`. The following list enumerates Boolean states which require an object ID as attribute and which can therefore be used as condition of `ObjectState`:

`InsideField #<n>` where `n` is the number of a `Field` primitive

`IsLoitering #<n>` where `n` is the number of a `Loitering` primitive

The following object properties and non-Boolean states are also available within the definition of `ObjectState`:

`ObjectsInState #<n>` where `n` is the number of an object state

`ObjectsInField #<n>` where `n` is the number of a `Field` primitive

`ObjectsOnScreen`

`Direction`

`Velocity`

`AspectRatio`

`ObjectSize`

`RelativeObjectSize`

Example

The following VCA Task script detects objects which are speeding within a specified field.

```
Field #1 := {…};
external ObjectState #1 :=
  Velocity within(30,*) and InsideField #1;
```

4.10 User-defined events

In the previous sections, several events have been introduced which are automatically generated together with the corresponding primitives. The user can define new events by using temporal relations between events. Besides temporal relations, additional conditions can be formulated in order to constrain events further. With these conditions, the user has access to event attributes and can formulate constraints for them. In the subsequent sections, the different possibilities are described in more detail.

4.10.1 Basic syntax

User events are defined in the following way:

```
[external] Event #<n> := <event>;
```

Thereby, `<event>` is a placeholder for any predefined event (like `EnteredField` as introduced in the previous sections or a user-defined event) or more complex event expressions as described in the upcoming subsections. The user can specify up to 32 events (from `1` to `32`) via the number `<n>`, but only the first 16 events (`<n>` from `1` to `16`) can be `external`. The same user event cannot be defined twice. `Event #<n>` inherits the attribute list from `<event>`, i.e. it provides the same list of attributes as `<event>`.
When the Alarm Task engine detects an external user-defined event, the corresponding alarm flag is set for exactly 1 processed frame. If the same alarm flag is used by several external rules (e.g. external `ObjectState` or external `SimpleState`), the alarm flag is set if any of the rules is active.
The following table summarizes the predefined events which can be used as `<event>` placeholder:

`FollowedRoute #<n>` where `n` is the number of a `Route` primitive

`CrossedLine #<n>` where `n` is the number of a `Line` primitive

`EnteredField #<n>` where `n` is the number of a `Field` primitive

`LeftField #<n>` where `n` is the number of a `Field` primitive

`Appeared`

`Disappeared`

`Idled`

`Removed`

4.10.2 Temporal relations

The most interesting operations on events are temporal relations. The `before` keyword combines two events in the following way. If the second event is triggered and the first event has been triggered before, another event with the same attributes as the second one is triggered. In its simplest form the syntax is as follows:

```
<event> := { <event> before <event> }
```

The curly brackets are mandatory to avoid ambiguous associations of nested constructions. With the following extension, a time interval can be specified which limits the chronology of the two events further.

```
<event> := { <event> before(<from>,<to>) <event> }
```

In this formulation, the first event is only a candidate for the second event if it has occurred at least `<from>` seconds and at most `<to>` seconds before the second event. Replacing `<to>` by the `*` (asterisk) symbol, an infinite time interval can be defined starting with `<from>`.

Adding the `not` keyword, it is even possible to check whether the first event has not occurred before the second event in the specified time interval (the time interval is again optional):

```
<event> := { <event> not before(<from>,<to>) <event> }
```

The `or` keyword can be used to trigger an event if either event A or event B has happened. The attribute list of both events must be the same:

```
<event> := <event> or <event>
```

In this case curly brackets can be omitted.

Example
`Event #1` marks objects which cross 1 of 2 lines. `Event #2` detects a pair of objects where 1 object passed the first `Line #1` and 1 object passed the second `Line #2` after at most 5 seconds. `Event #3` is triggered by objects which pass `Line #2` while `Line #1` was not crossed for 5 seconds.

```
Line #1 := {…};
Line #2 := {…};
external Event #1 := CrossedLine #1 or CrossedLine #2;
external Event #2 := {
   CrossedLine #1 before(0,5) CrossedLine #2
};
external Event #3 := {
   CrossedLine #1 not before(0,5) CrossedLine #2
};
```

4.10.3 Conditions

Often it is necessary to constrain events by their attributes or by properties of involved objects. In the VCA Task script language, this is supported via the `where` clause.

```
<event> := { <event> where <condition> }
```

The Alarm Task engine will only trigger if the `<condition>` is satisfied at the time when the `<event>` occurred. Which states and properties can be used in the `<condition>` clause depends on the attribute list of `<event>`. In principle, events with an empty attribute list can have a `<condition>` clause similar to the one of `SimpleState`. If the attribute list contains `OID:oid` as attribute, the `<condition>` clause is similar to the one of `ObjectState`.
With the concepts introduced so far it is possible to detect whether the same object has passed first 1 line and afterwards another line. The VCA Task script language solves this task with a combination of `before` and `where` keywords.

```
<event> := { <event> before <event> where <condition> }
```

In the `<condition>` clause following the `where` keyword it is possible to access attributes of both events involved in the `before` relation. An attribute `<x>` of the event to the left can be accessed via `first.<x>` whereas attributes of the event to the right are available via `second.<x>`. If an attribute `<x>` belongs exclusively to 1 of the 2 events, the specification of `first` and `second` is optional. If an attribute `<x>` belongs to both events, the attribute is associated to the `second` event, which is the more recent one. Any of the other `before` variants described in subsection "Temporal relations" can be similarly combined with a `<condition>` clause.

Example

With these extensions, objects can be found which cross two lines in a row at a certain speed, as shown by `Event #1`.

```
Line #1 := {…};
Line #2 := {…};
external Event #1 := {
  CrossedLine #1 before(0,5) CrossedLine #2
  where first.oid == second.oid and
    Velocity(second.oid) within(30,*)
};
```

4.10.4 State changes

In order to detect e.g. changes of an object's appearance, the keywords `OnChange`, `OnSet`, and `OnClear` are introduced. They can be combined with any user-defined state and trigger an event if the state changes its value, if the state becomes `true`, or if the state becomes `false`. The syntax is as follows:

```
<event> := OnChange <state>
<event> := OnSet <state>
<event> := OnClear <state>
```

Thereby, `<state>` is either a previously defined `ObjectState #<n>` or `SimpleState #<n>`. The attribute list of the `<state>` is passed to the corresponding change event. For instance, the attribute list of an `OnChange SimpleState #<n>` event is empty, whereas the attribute list of an `OnSet ObjectState #<n>` has `OID:oid` as its only attribute.

Example

The following VCA Task script detects objects which are changing their shape from tall and thin to flat and wide.

```
ObjectState #1 := AspectRatio within(1.2,*);
ObjectState #2 := AspectRatio within(*,0.8);
external Event #1 := {
  OnClear ObjectState #1 before OnSet ObjectState #2
  where first.oid == second.oid
};
```

`ObjectState #1` is `true` if the object is taller than wide. `ObjectState #2` is `true` if the object is wider than tall. For objects which are almost square, both states are `false`. The task is to look for objects whose `ObjectState #1` was set some time ago and whose `ObjectState #2` is set now. Hence, it is sufficient to wait for `OnSet ObjectState #2` and check if `OnClear ObjectState #1` has happened before.

4.10.5 Temporary states

In order to have an event triggering a temporary state, the keyword `within` is introduced. It is added after an event to delay an alarm, to extend an alarm state, or to temporarily combine with other states. The syntax is as follows:

```
<state> := <event> within(min,max)
```

Thereby, `<event>` is either a previously defined `Event #<n>` or an arbitraty predefined event like `EnteringField`, `Appeared`, or `CrossingLine`. The attribute list of the `<state>` is passed to the corresponding change event. For instance, the attribute list of an `OnChange SimpleState #<n>` event is empty, whereas the attribute list of an `OnSet ObjectState #<n>` has `OID:oid` as its only attribute.

Example
The following VCA Task script triggers an alarm once an object appears. The alarm lasts for 30 seconds , even if the object disappears in the meantime.

```
external SimpleState #1 := Appeared within(0,30);
```

The next line delays the triggering of the alarm for 10 seconds after the object appeared. If the object disappears in the meantime, no alarm is triggered.

```
external ObjectState #2 := Appeared within(10,*);
```

An `ObjectState` is used here, because an object state depends on the object and is `false` if the object is gone. A `SimpleState` is staying `true` until the algorithm is reset, for example due to reconfiguration.
The next example is more complex. It shows how to combine the temporary state with other states. First a `Counter #3` is defined that counts objects in an arbitrary object state. If no objects are in this state, the `SimpleState #31` is `true`. The resulting state `external SimpleState #3` is the `SimpleState #31` delayed on activation by 5 seconds.

```
ObjectState #32 :=...;
Counter #3 := { ObjectsInState #32 };
SimpleState #31 := Counter #3 within (0,0);
external SimpleState #3 := SimpleState #31 and !(OnChange SimpleState
#31 within(0,5));
```

Bosch Security Systems B.V.
Torenallee 49
5617 BA Eindhoven
Netherlands
www.boschsecurity.com