VCA Task Script Language Specification

Table of Contents

1 Definitions

1.1 Events & States

Whereas states are temporal functions, events occur each time a state changes its value. The query 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.

1.2 Properties

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

1.3 Rules

When speaking about rules, both - events and states - are meant which are output by the rule 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 from the frame on where the event has been triggered to the next processed frame where the rule is inactive again.

1.4 Rule Engine

The rule engine parses the output of the video content analysis module and searches within this stream for rules defined according to the VCA Task Script Language.

1.5 Method of describing syntax

The following elements `<>` , `[]` , `OR` , and `|` are not part of the VCA task script language syntax. These elements are used to describe classes of syntax elements (e.g. `<n>` for an number which needs to be defined), optional elements (e.g. `[DebounceTime(4)]` for element which can be omitted) and alternative elements (e.g. `(HasClass|HadClass) OR HasColor`). A repetition of syntax elements are signalized with dots `...` (e.g. `Point(<x>,<y>) ... Point(<x>,<y>)`).

1.6 Predefined syntax elements

`<i>` := An unique internal ID of an event, state, loitering, color histogram, field or counter which is a integer number from 1 to 32.

`<n>` := An unique ID of an external event, external state, or Line which is a integer number from 1 to 16.

`<j>` := An unique ID of an route which is a integer number from 1 to 8.

`<x>` := An x-coordinate of the screen beginning from the left side. The range is defined by the `Resolution` exp ression. It is a floating point number with 3 decimal places.

`<y>` := An y-coordinate of the screen beginning from the top. The range is defined by the `Resolution` expression. It is a floating point number with 3 decimal places.

2 System Integration

2.1 System Overview

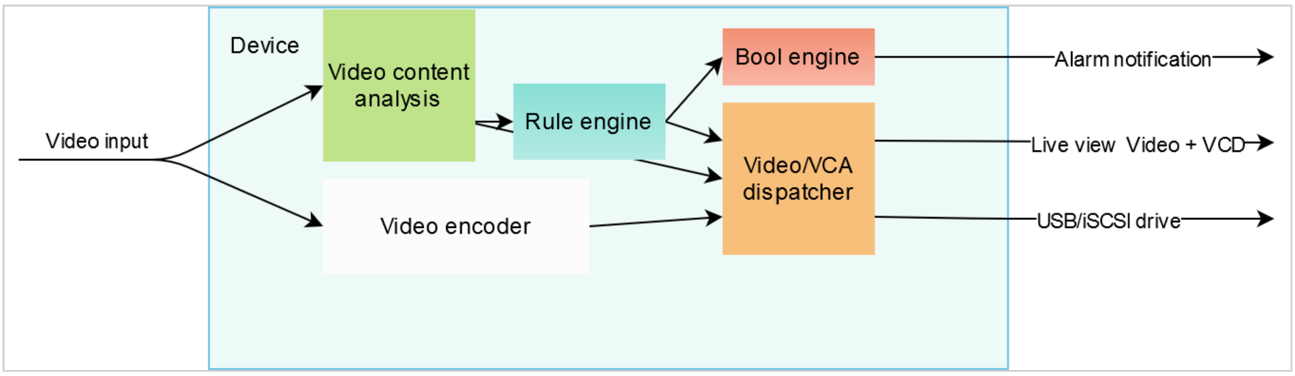

Figure 1: System overview of device

Figure 1 shows the parts of the architecture which are relevant for video content analysis. First, the video signal is sent to the video encoder and the video content analysis software. The former outputs e.g. H263 or H264 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 video content analysis 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 rule engine which does the detection of user defined rules. The output of the rule engine are up to 16 alarm flags. On the one hand, these are merged into the VCD stream. On the other hand, the bool engine is notified about changes of alarm flags. The bool engine allows association any kind of actions, like sending emails, triggering a relay output, notification of a video management system, etc., to alarm flags.

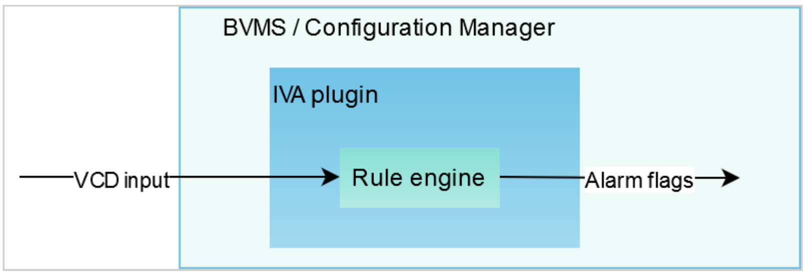

Figure 2: System overview of plugin

The rule engine is configured with a script specified in the VCA Task Script Language. The same language is used in Bosch’s forensic search applications (see Figure 2) to look for unforeseen events in recordings. There, the recorded VCD streams are fed through another instance of the rule 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 rule engine is packed into a plugin which can be easily exchanged. The IVA plugin is the first one supporting the VCA Task Script Language and configuration of the corresponding rule engine.

2.2 User-defined Tasks

Via the IVA plugin the rule engine can be configured. To simplify matters, the plugin provides a comprehensive list of predefined queries with corresponding wizards. The wizards enter automatically query code into the rule 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 IVA plugin enables the definition of up to 16 tasks corresponding to the first 16 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 (see Predefined syntax elements). The name of the task can be specified within the quotation marks.

The number after T: indicates the task type, which should be set to 0 for user-defined tasks.

(see predefinded tasks types)

Only use user-defined task type 0 for your self-written tasks.

Supported task types:

task type	type number	comment
User-defined	0	
Object in field	1	
Crossing lines	2	
Loitering	3	
Condition change	5	
Following route	6	
Tampering	7	
Remove object	8	
Idle object	9	
Entering field	10	

task type	type number	comment
Leaving field	11	
Similarity search	13	
Crowd detection	14	
~~Counter~~	~~15~~	obsolete with firmware version 6.30
~~People counter~~	~~16~~	obsolete with firmware version 6.10
Line counter	17	
Occupancy counter	18	
~~Flow in field~~	~~40~~	task for **IVA Flow** algorithm, obsolete for IVA pro
~~Counter flow in field~~	~~41~~	task for **IVA Flow** algorithm, obsolete for IVA pro
Motion in field	42	task for **MOTION+** algorithm
Motion region detector	43	**Onvif** only generated task

3 Syntax

3.1 Types

3.1.1 Basic Types

The query 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 degree, 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[1] within the specified range.

There exists a special value `NAN` (not a number) which indicates that the value does not exist. For instance, when the direction of an object is accessed which does no longer exist, the return value is `NAN` .

Example (Speeding)

The subsequent query 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 function Velocity 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.

3.1.2 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 query 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 one 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
};
```

3.2 Comments

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

3.2.1 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 */
```

3.3 Script normalization

The query language provides method how the following coordinates are normalized within the given max and min values. The keyword Resolution should be placed 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 of the image.

3.3.1 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 )
};
```

3.4 Primitives

The query 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 rule-engine is running. In the firmware this limits the number of trajectories to 8, and the total number of primitives to 32.

3.4.1 Field

Syntax

Field primitives are defined in the following way:

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

`<i>` is the number of the area 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 IVA 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 area, 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 an area.

If a flow detector is used the DebounceTime has a different meaning. The `<time>` specifies the post-alarm time for the area. This ensure that if many short flow alarms are detected within the debounce time `<time>` they are merge 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, where BaryCenter is the 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 only of a single point.

States

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

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

```
ObjectsInField #<i> -> INTEGER
```

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

Events

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

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

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

The event `LeftField` is triggered, when an object leaves `Field #<i>`. The object which has caused the event can be queried by the attribute oid.

One should note that an object which is first detected within an area does not trigger the corresponding `EnteredField #<i>` event.

Example

The following example triggers an alarm if the same object has first entered the specified area 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
};
```

3.4.2 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 must have between 2 and 16 points, whose coordinates are specified in pixels with the image resolution processed by the IVA 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 former 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 are considered to trigger a line crossing. Possible trigger points are `BaryCenter` or `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 Example "Crossing Two Lines".

3.4.3 Route

Syntax

Route primitives are defined in the following way:

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

`<j>` is the number of the Route and must be between 1 and 8. A Route has at least 2 and at most 8 points. The coordinates of the points are specified in pixels with the image resolution processed by the IVA 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 #<j>`. 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 gap 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`.

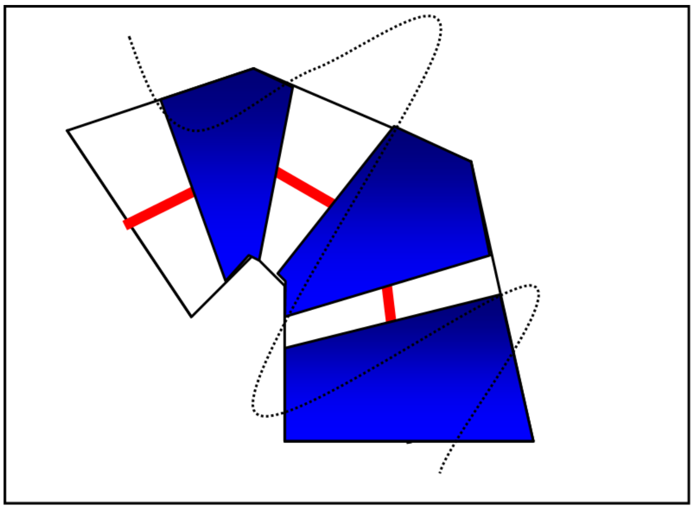

Figure 3. This illustrates an object following a predefined route. The parts of the trajectories stripe which have been visited by the object are marked in blue. The gaps in between are highlighted with a red line.

Events

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

The event `FollowedRoute #<j>` is triggered, when an object has followed the `Route #<j>`. 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;
```

3.4.4 Loitering

Syntax

Loitering primitives are defined in the following way:

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

`<i>` 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 calibration must have been calibrated beforehand.

States

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

The state `IsLoitering #<i>` of an object oid is set, while 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;
```

3.4.5 ColorHistogram

Syntax

ColorHistogram primitives are defined in the following way:

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

`<i>` is the number of the color histogram primitive and must be between 1 and 32. The color histogram 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, where `<h>` (a value between 0 and 360) represents the hue component, `<s>` (a value between 0 and 100) the saturation, and `<v>` (a value between 0 and 100) the intensity. Figure 4 visualizes the three components of the HSV color space.

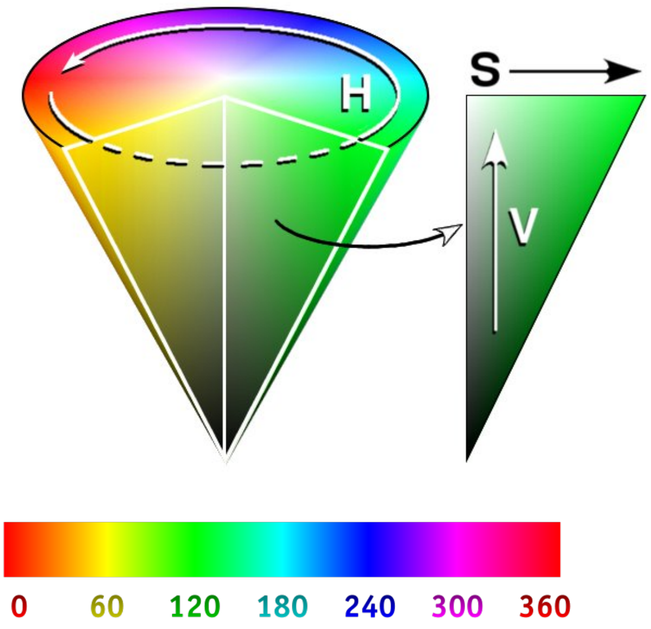

Figure 4. HSV cone

`<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% don't care.

States

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

The state `SimilarToColor #<i>` of an object oid is set, while 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;
```

3.4.6 FlowDetector

Syntax

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

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

OR

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

The flow detector operates on the significant flow field computed by the IVA 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 two directions are specified, a motion vector must pass at least one for further processing. Each directional filter consists of a `<minangle>` and `<maxangle>` specified in degree. The coordinate system for angles is shown in Figure 5.

In the second definition, the direction is automatically determined based on a main flow analysis. The last `< time window>` seconds are considered in order to compute the main flow direction. As soon as a dominant direction could be 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> degree 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. 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 select an activation interval within which the flow detector fires 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 #<i> -> BOOLEAN
```

The state DetectedFlow #<i> is set, while there is a significant flow fulfilling all the constraints defined in the corresponding FlowDetector #<i> 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;
```

3.4.7 CrowdDensityEstimator

Syntax

CrowdDensityEstimator primitives must be 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 detected the crowd in front of the background reference. This scene is limited by the CrowdDensityField which must be specified in advance. The IVA 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 no alarm will be set if the crowd level will be within the activity thresholds at least the specified period of time. Else if SmoothingTime is used the average over the specifed period must be within the activity thresholds to trigger an alarm.

States

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

The state `EstimatedCrowdDensity #<n>` is set, while there is a significant crowd level fulfilling all the constraints which has been defined in the corresponding `CrowdDensityEstimator #<n>` definition.

Example

The following example triggers an alarm, if the crowd level is at least 10s greater or equal to 25% within the specified crowd field which 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;
```

3.4.8 MotionDetector

Syntax

MotionDetector primitives must be 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 activation level can be limited by setting the Activity interval <minactivity> and <maxactivity>. The activity is specified in percent of the selected area. Furthermore the trigger activation level can by limited by the size of the connected cells (Size). This interval (<minsize>,<maxsize>) is specified in percent of the whole screen.

States

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

The state DetectedMotion #<n> is set, while there is a significant motion level fulfilling all the constraints which has been defined in the corresponding MotionDetector #<n> definition.

Example

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

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

3.4.9 SoundDetector

Syntax

SoundDetector primitives must be defined in the following way:

```
SoundDetector #<n> := {
  <classnames>
  [MinConfidence(<confidence>)]
  [Direction(<min>,<max>)]
};
```

The SoundDetector triggers an alarm if an sound alarm event fulfills the defined criteria.

The SoundDetector must be provided with a class, subclass audio. The minimum confidence is optional and is in the range of 0..100. Also optional is a direction filter which evaluates the provided direction of arrival (DOA). Note: Direction 0 with 0 tolerance means no direction filter is applied.

```
DetectedSound #<n> -> BOOLEAN
```

The state DetectedSound #<n> is set, when the corresponding SoundDetector #<n> is true.

Example

The following example triggers an alarm, if a gunshot has been detected with a minimum confidence of 40 and DOA is between -45 and 45 degree.

```
SoundDetector #1 := { Gunshot MinConfidence(40) Direction(-45, 45) };
external SimpleState #1 := DetectedSound #1;
```

3.5 Object-Specific Events

The main task of the IVA 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.

Merged, Idled, and Removed are special cases of disappearing objects with the following meaning. If two objects come too close to each other, one of the two becomes partially hidden by the other. This situation is indicated by a Merged event. 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, somewhere. Similarly, an object can be picked up by a person triggering a

Removed event. A special case of an appearing object is when it was previously hidden by another object. E.g. if two pedestrians pass each other, one can become hidden during the passing and reappears after the passing. This reappearing is indicated by a Split event, the counterpart of the Merged event.

3.5.1 Counter

Syntax

Counters can be defined in one of the following ways:

```
[external] Counter #<n> := { Event #<e> [Text(“<string>”)] [TopLeft(<x>,<y>)]
[Mode(KeepMax|Wraparound) within(<min>,<max>)] };
OR
[external] Counter #<n> := { Counter #<c> + Counter #<k> [Text(“<string>”)]
[TopLeft(<x>,<y>)] [Mode(KeepMax|Wraparound) within(<min>,<max>)] };
OR
[external] Counter #<n> := { Counter #<c> - Counter #<k> [Text(“<string>”)]
[TopLeft(<x>,<y>)] [Mode(KeepMax|Wraparound)within(<min>,<max>)] };
OR
[external] Counter #<n> := { <num-expr> [Text(“<string>”)] [TopLeft(<x>,<y>)]
[Mode(KeepMax|Wraparound) within(<min>,<max>)] };
```

`<e>` , `<c>` , `<k>` stands for event or counter numbers which are previously defined, `n` is the number of the counter defined now.

`<string>` is a maximal 32 byte utf-8-encoded string.

`<x>` , `<y>` are the x and y coordinates in the resolution which is defined by `Resolution` (see Script Normalization).

`<min>` , `<max>` are the minimum and maximum counter values which are displayed. $\in \mathbb{N}_0$

The Counter can either count the number of triggered event 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 internally used 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 the mode `KeepMax` is set, the counter remains at the upper bound. In the letter mode `Wraparound` the counter starts again with the lower bound (e. g. 0). The minimum and maximum bound can be set with the argument within (<min>, <max>), in which the lower bound corresponds to <min> and the upper bound to <max>. The position of the displayed counter value 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 exemplarily shown:

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

Therefore, an alarm is fired 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,*);
```

3.5.2 Object Events

Object specific events can be triggered by appeared and disappeared objects.

Events

The following events are triggered if an object appears, disappears, stands still, or starts moving.

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

Example

```
external Event #1 := Idled;
```

3.6 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.

3.6.1 Functions

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

This function returns the current direction in degree 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 0 degree, from the top border to the bottom border corresponds to 90 degrees, from the left border to the right border corresponds to 180 degrees, and from the bottom border to the top border corresponds to 270 degrees (see Figure 5).

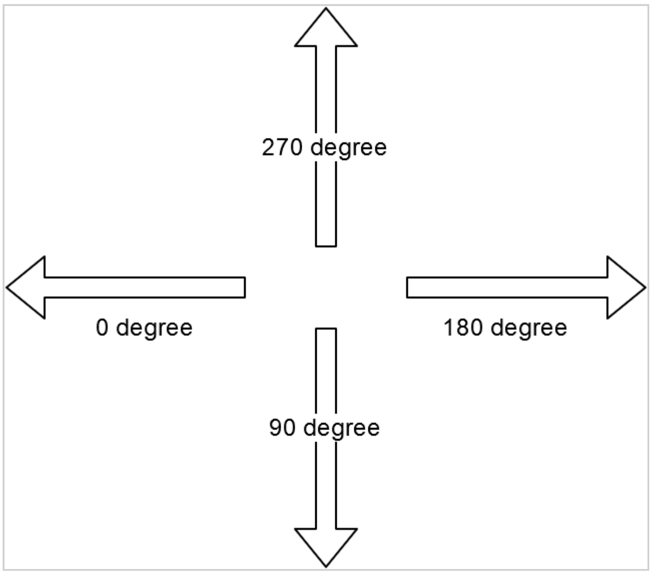

Figure 5: Directions

```
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 6).

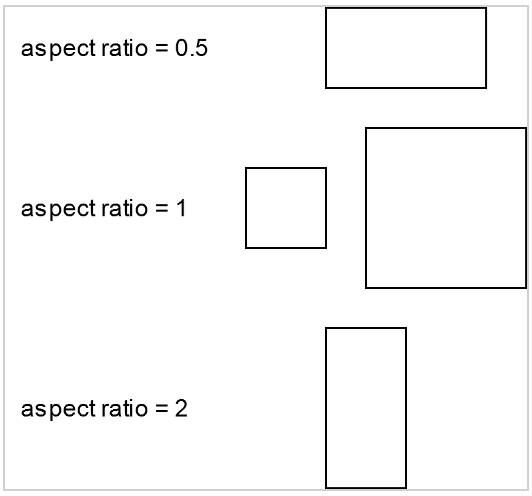

Figure 6: Aspect ratio definition

```
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.

```
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.

3.6.2 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.

```
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.

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

returns true, if a class of the specified class names is detected at the object in the current frame. If the object is not present in the current frame, the return value is false.

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

returns true, if a class of the specified class names has been detected at the object since the last appearance of the object. If the object is not present in the current frame, the return value is false.

<classnames> is a list of following class names separated by empty space:

- `Person, a walking person (or standing person available for 2D/3D traffic tracking mode)`
- `Vehicle, a vehicle (e.g. one of the following Bike, Car, Truck, ...) available for 2D/3D traffic tracking mode`
- `Bike, a bicycling bike or motorcycle`
- `Bicycle, a bicycle available for 2D/3D traffic tracking mode`
- `Motorcycle, a motorcycle available for 2D/3D traffic tracking mode`
- `Car, a driving car ( or standing car available for 2D/3D traffic tracking mode)`
- `Truck, driving big car, truck or bus (or standing truck or bus available for 2D/3D traffic tracking mode)`
- `Bus, a bus available for 2D/3D traffic tracking mode`
- `CT, class generated by Camera Trainer`
- `Weapon, a long gun or short gun carried by a person (available for visual gun detection mode)`
- `NoClass, none of the above classes`

```
HasSubclass ( CT, <subclassid> ) -> BOOLEAN
```

returns true, if a subclass of the specified detector (currently only CT supported) is detected at the object in the current frame. If the object is not present in the current frame, the return value is false. This is introduced in firmware 7.60. The <subclassid> is the number shown in Camera Trainer tab or on the class flag. (A synonym introduced in firmware 7.10 is SOD, which is deprecated.)

3.7 Tamper States

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

```
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.

```
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.

```
SignalLoss -> BOOLEAN
```

If the video signal is lost, the SignalLoss state is set.

```
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 IVA algorithm and the reference checking of the IVA algorithm has been enabled, IVA 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 query language via the RefImageCheckFailed state.

3.7.1 Example

The following query triggers an alarm on any detected tamper action.

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

One should 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.

3.8 User-defined States

User-defined states are always Boolean states, i.e. these states take either the value true or false. The user-defined states have either no attributes or the object ID has attribute. The former states are called SimpleStates, whereas the latter ones are called ObjectStates. In the subsequent sections their syntax and usage is explained in more detail.

3.8.1 SimpleStates

SimpleStates are defined in the following way:

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

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 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 SimpleStates. The <stateoptions> are introduced with firmware 7.60. Possible Options are `DebounceTime(<time>)` and `AggregationTime(<time>)`. The DebounceTime specifies the minimum time that a SimpleState must be set before it triggers an alarm. If an alarm is triggered, the reset of the alarm is delayed by the time specified as AggregationTime. If the SimpleState is set again while AggregationTime countdown runs the reset is stopped and the delay starts again on next unset of the SimpleState. See also Figure 7.

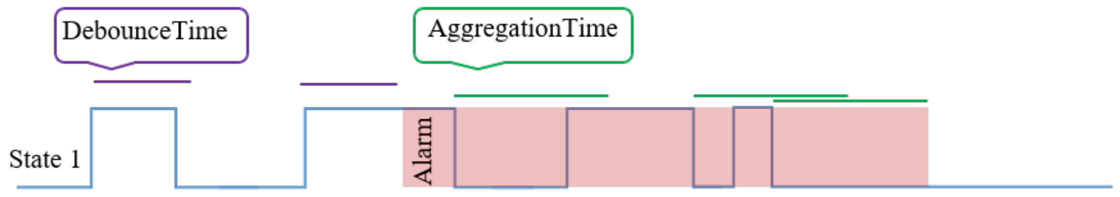

Figure 7: Debounce Time and Aggregation Time of a SimpleState

The following table summarizes the predefined states, which can be used instead of the `<condition>` placeholder besides other previously defined SimpleStates:

SignalTooNoisy
SignalTooDark
SignalTooBright
SignalLoss
GlobalChange
RefImageCheckFailed
CameraMoved (added with firmware 7.60)

Example

The following query triggers an alarm if a movement of the camera was detected for 3 seconds it is set at least 10 seconds even if there is a flickering of the state in between:

```
external SimpleState #1 := CameraMoved with DebounceTime(3) AggregationTime(10);
```

3.8.2 Boolean Composition of Conditions

Several conditions can be composited. 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 last disjunctions. Furthermore, the priority can be controlled by embracing sub-expressions with brackets:

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

3.8.3 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 one of the two bounds is replaced by an asterix `*`, 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 equal operation 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, in a field or which trigger an object state.

One should note that the conditions described in this section return false, if the value of one `<num-expr>` is NAN.

Example

The following query triggers an alarm if at least 2 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 at most 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
};
```

3.8.4 ObjectStates

ObjectStates are defined in the following way:

```
[external] ObjectState #<n> := <condition> [with DebounceTime(<time>)];
```

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 can be external. The same ObjectState cannot be defined twice.

In contrast to `SimpleStates` , which are instantiated only once, `ObjectStates` are 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
```

The DebounceTime is introduced with firmware 7.60. In contrast to `SimpleStates` DebounceTime the time is evaluated for each object. The ObjectState is triggered if the object remains in the state for at least the given DebounceTime. See also Figure 8.

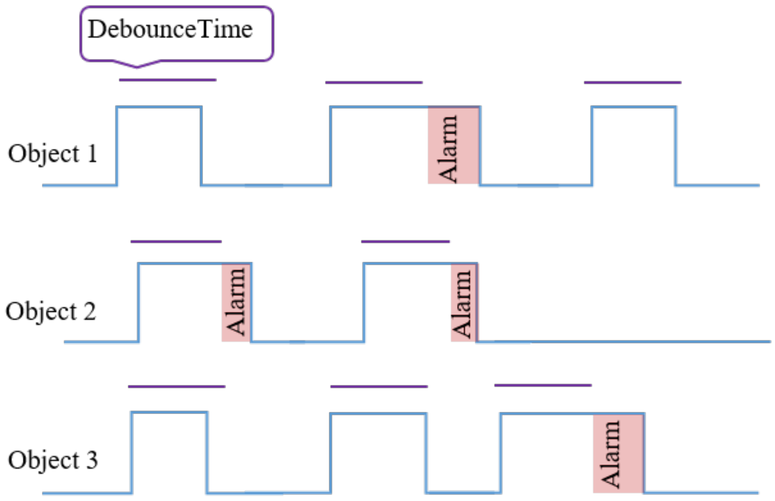

Figure 8: DebounceTime of ObjectState

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

InsideField #<n>	where n is the number of an area 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 ObjectStates:

ObjectsInState #<n>	where n is the number of an object state.
ObjectsInField #<n>	where n is the number of an area primitive.
ObjectsOnScreen	
Direction	

Velocity	
AspectRatio	
ObjectSize RelativeObjectSize	

Example

The following query detects objects, which are speeding within a specified area.

```
Field #1 := {…};

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

The following query detects objects, which have an upright aspect ratio for at least 5 seconds.

```
external ObjectState #2 := (AspectRatio within (2,*)) with DebounceTime(5);
```

3.9 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.

3.9.1 Basic Syntax

User events are defined in the following way:

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

Thereby, `<event>` is a placeholder for either predefined events (like EnteredField as introduced in the previous sections), another already defined 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 rule engine detects an external user-defined event, the corresponding alarm flag is set for exactly one processed frame. If the same alarm flag is used by several external rules (e.g. external ObjectStates or external SimpleStates), 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 #<j>	where j is the id of a Route primitive.
CrossedLine #<n>	where n is the id of a line primitive.
EnteredField #<i>	where i is the id of an area primitive.
LeftField #<i>	where i is the id of an area primitive.
Appeared	
Disappeared	
Idled	
Removed	
Merged	
Split	

3.9.2 Temporal Relations

The most interesting operations on events are temporal relations. The before keyword combines two `<event>`s 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 `*` 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 one of two lines. `Event #2` detects a pair of objects where one object passed the first `Line #1` and one 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 := { Point( 10, 10 ) Point( 10, 50 );
Line #2 := { Point( 50, 50 ) Point( 50, 10 );
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 };
```

3.9.3 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 rule 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 empty attribute list can have a `<condition>` clause similar to the one of SimpleStates. If the attribute list contains OID:oid as attribute, the `<condition>` clause is similar to the one of ObjectStates.

With the so-far introduced concepts, it would be possible to detect whether the same object has passed first one 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 left event can be accessed via first. `<x>` whereas attributes of the right event are available via second `<x>`. If an attribute `<x>` belongs exclusively to one of the two 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`. `Event #2` detects if two persons were passing a gate close to each other and were separating on the other side. For robustness, the size of the objects is checked.

```
Line #1 := { Point( 10, 10 ) Point( 10, 50 );
Line #2 := { Point( 50, 50 ) Point( 50, 10 );
external Event #1 := { CrossedLine #1 before(0,5) CrossedLine #2
                       where first.oid == second.oid and
                             Velocity( second.oid ) within (30,*)
};

external Event #2 := { CrossedLine #1 before(0,5) Split
                       where first.oid == second.oid and
                             ObjectSize( second.oid2 ) within (1,*) and
                             ObjectSize( second.oid ) within (1,*)
};
```

3.9.4 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 either 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 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 query 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 quadratic both states are false. The task is to look for objects whose `ObjectState #1` was set some time ago and whose `ObjectState #2` is now set. Hence, it is sufficient to wait for `OnSet ObjectState #2` and check if `OnClear ObjectState #1` has happened before.

3.9.5 Temporary States

In order to create temporary states triggered by an event the keyword within is used after an event. This can be used to delay an alarm, to extend an alarm state or to temporary 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 arbitrary predefined event e.g. `EnteringField`, `Appeared` or `CrossingLine`. The attribute list of the `<state>` is passed to 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.

Examples

The following query alarms for 30s if an object appears, even if the object vanished meanwhile.

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

The next line delays the alarm after object appears for 10s. If the object vanishes meanwhile no alarm is triggered.

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

An object state `ObjectState` depends on the object and is false if the object is gone, were as a `SimpleState` would stay true till algorithm is reset, e.g. on reconfigure. Therefore, an `ObjectState` is used here.

The next example is a more complex one, which shows how the temporary state can be combined with other states. First a `Counter #3` is defined which 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 := ObjectOnScreen;
Counter #3:= { ObjectsInState #32 };
SimpleState #31:= Counter #3 within(0,0);
external SimpleState #3:= SimpleState #31 and !(OnChange SimpleState #31 within(0,5))
;
```

4 Change History

Revision	**Date**	**Comment**
1.14	2023-08-16	Description for Task Script Language for firmware 9.0 • added class Weapon
1.13	2022-12-02	Description for Task Script Language for firmware 8.80 • additional AudioAIDetector
1.12	2021-08-20	Description for Task Script Language for firmware 8.1 • additional classes added
1.11	2019-11-13	Description for Task Script Language for firmware 7.6 • DebounceTime is added to SimpleState and ObjectState and AggregationTime is added to ObjectState • class CT is introduced
1.10	2019-04-01	Description for Task Script Language for firmware 7.1 • a state can be used to generate temporary state • class SOD is introduced
1.9	2018-11-02	Description for Task Script Language for firmware 6.6 • an event can be used to generate temporary state
1.8	2016-05-23	Description for Task Script Language for firmware 6.3 • extended ObjectSet by FootPoint • added TriggerPoint to Line and Route • extended number of allowed Points in Line • added RelativeObjectSize in section “Object Properties” • added ObjectInState non-Boolean state • extended Counter Description
1.7		Description for Task Script Language for firmware 6.0 • added definition of HasClass and HadClass • added definition of MotionDetector
1.6	2013-02-26	Description for Task Script Language for firmware 5.6 • added definition of Resolution

1.5	2012-04-11	Description for Task Script Language for firmware 5.5 • added definition of Counter
1.4	2012-01-30	Description for Task Script Language for firmware 5.0 • added definition of CrowdDensityEstimator • added simple state EstimatedCrowdDensity
1.2	23.04.2008	Description for Task Script Language for firmware 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
1.1	16.10.2007	Description for Task Script Language for firmware 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
1.0	15.10.2007	Description for Task Script Language for firmware 3.0