Roomle Configuration Catalog Structure Format

February 25, 2019
Kernelversion 2.11.0

Roomle GmbH
Peter-Behrens-Platz 2 | 4020 Linz | Austria
www.roomle.com

Contents

Chapter 1

Definitions

1.1 Catalogs

A Catalog consists of items, components, tags, materials. It can be private (visible only to selected users and the owner) or public (available for all Roomle users).

1.2 Tag

Tags are the way items are organized in catalogs. Each catalog must have at least one root-tag. Tags can be structured hierarchically and one tag can be a child of multiple other tags. Each Tag is linked to one catalog and must have an (globally) unique identifier.

1.3 Item

An item is an object which can be added to the plan. This can be a normal 3D object or a pre-defined configuration of components. Each item belongs to exactly one catalog but can be linked to multiple tags. Items which are not linked to any tag are not visible to the user. Every item must have an (within the Catalog) unique SKU.

1.4 Component

A component is the basic element used for configurable objects. Each component contains a definition on how this component can be configured and used within configurations. Similiar to items, components can be linked to multiple tags. Every component must have an (within the catalog) unique identifier.

1.5 Mesh

Within each catalog you can define meshes to be used in the geometry-script of a component. Each mesh can have data for different formats and qualitylevels. At the moment only crt (corto compressed meshes) with Realtime quality (level 50) is used. Within the script the mesh can be inserted with the AddExternalMesh command. The MeshId is the combined id <catalogId>:<meshId>.

Defined meshes (external meshes in the script) improve loading performance since only those meshes are loaded that are currently needed. In contrast to scripted meshes where the whole mesh is part of the script and needs to be interpreted even if not shown. Furthermore defined meshes provide the client with the ability to reuse the mesh (since defined meshes have an id and are const by definition) within the scene which also improves performance and memory usage.

1.6 Material

Within each catalog you can define materials to be used in the geometry-script of a component. Each material can have an assigned texture to be used. Every Material must have an (within the Catalog) unique identifier.

1.7 Configurations

When talking about configurable items you must distinguish between what we call "configuring variants" (which is basically just changing colors, product dimensions, etc.) and "combining components to one object" (creation of new objects and products - real configuration). Components can be seen as templates for the basic elements of configurable objects. A component itself can not be used within a plan. To use a component within a plan it must be embedded into a configurated item, which is an item with a given configuration. The configuration (see section configuration format) defines the actual variant of the involved components used in the configuration.

1.7.1 Configuring variants

An example for configuring variants is a table where you can change the dimensions and the material of the surface. In Roomle this can be modeled as one component with parameters for the dimensions and the material. This table within a plan is defined through a configuration containing only one component.

1.7.2 Combining Components

More complex configurations arise when multiple components can be combined together, e.g. a frame with different possible shelves that can be added. Every configuration must have

exactly one root-component. Details on docking components can be found in 2.5

Chapter 2

Roomle Configuration Format

The Roomle Configuration Format is serialized as JSON file.

2.1 Coordinate system

The coordinate system within the Roomle ConfigurationScript is a left-handed cartesian coordinate system. All points and sizes must be stated in mm.

2.2 Component definition

The component definition defines how a component should be displayed, which parameters are possible and the interaction with other components. It includes all information needed within the configurator. This includes:

- A condition when the component is considered valid. This can be used for Components who must have specific parameters/dockings set to be valid.
- A list of possible parameters including the possible values and a default value per parameter
- A list of possible ParentDockings and ChildDockings. These are the connectors to combine different components.
- A list of possible SiblingPoints. These are connectors throu which components can transfer data (e.g. parameter values) regardless of the parent-child connection.
- A list of possible AddOnSpots. These are visual aids for the User.
- A list of possible Subcomponents which can be used within the geometry script and will be displayed in the partlist
- The article number script written in RoomleScript

- The geometry script written in RoomleScript
- The geometryHD script written in RoomleScript. This version is used in special clients for higher resolution and quality. Besides that it follows the same logic and definitions as the geometry script. Everything that can be done in the geometry script can be done in geometryHD too.
- The previewGeometry script written in RoomleScript. When provided, this geometry is used to preview objects during adding of new children. If not provided, the geometry script is used. All geometry-script functions work the same in the previewGeometry. If provided the previewGeometry script should be less complex than the real geometry script to improve performance during the insertion.
- The packageSize contains a Array<int>, these are all the numbers of packages which are allowed for this component
- The packaging contains a list of sizes with conditions for adding certain sizes if needed
- The price calculation written in RoomleScript
- an onUpdate RoomleScript which is executed everytime something changes within the component. Within this script even parameters of the component may be changed.
- an onValueChange RoomleScript which can be provided for every Parameter and is executed on startup (change from no value to the first/default value) and everytime this parameter changes.

2.2.1 Parameters

A component is completly defined by the values of its parameters. Parameters have multiple ways to controll where and how it is shown and behaves.

- visible: shown to the user as part of the component
- enabled: user can modify the value
- global: all global parameters with the same key within a configuration are combined together and shown globally. The global value for a parameterkey may differ from the actual value on the component (if the component-parameter is changed after the global value is set). New components get the global value assigned automatically on dock.
- visibleAsGlobal: the global parameter is shown if any of the connected component-parameters is set as "visibleAsGlobal", otherwise its invisible globally. visible and visibleAsGlobal can be completly independent.
- visibleInPartlist: if this parameter should be shown in the partlist. parameters not visible in the partlist are also ignored in the aggregation of components in the partlist.

2.3 Subcomponents

Every component can contain multiple subcomponents. A subcomponent references a component with all of its scripts and computations. The main component may set parameters of the subcomponents via assignments. It's also possible for the main component to define one or more subcomponents as active. If defined, the main component can take parameters of the active subcomponents to supersede its own.

2.3.1 Supersedings

For active subcomponents the main component may define supersedings. If a superseding parameter is defined, it completly replaces the parameter of the main component with the same key if one exists. This means that from outside it behave as if the parameter were the parameter of the main component itself although the validValues and all calculations are done in the subcomponent.

This is specially useful for cases where the main component acts as a metacomponent which only decides which subcomponent is used. In this case all calculations can stay subcomponent specific without the need to copy them to the main component while still having all logic available.

The values of the superseded parameters are also available in the main component (and may override existing values in the main component) and can be used in geometry, docking etc. Be aware that the values of the superseding parameters may not be available on the initial executions of "onUpdate" since the component needs to initialize itself before knowing what parameters will be superseded. Consider using ifnull in such a case.

```
{
  "id": "sampleCatalog:component1",

  "labels": {
    "de": "deutsches Label",
    "en": "english Label", ...
  },
  "parameters": [
    {
      "key": "Length",
      "enabled": "true",
      "visible": "true",
      "visibleInPartList": "true",
      "unitType": "length",
      "labels": {
        "de":"Laenge",
        "en":"Length"
      },
      "type": "Decimal",
      "defaultValue": 1400,
```

```
      "validValues": [
        1400,
        1600, ...
      ]
    }, ...
  ],
  "pricing": [
    {
      "region": "RML_DEFAULT",
      "currency": "EUR",
      "price": "<price calculation skript>"
    }
  ],
  "possibleChildren": [  { "componentId":"sampleCatalog:component2" }
     ... ],
  "parentDockings": {
    "points": [ {
        "position": "{0,0,0}",
        "mask": "DockingMask1",
        "rotation": "{0,0,0}",
        "assignments": {}
      },...
],
    "lines": [{
        "position": "{(-Breite/2),0,-42.5}",
        "mask": "DockingMask2",
        "assignments": {},
        "rotation": "{0,-(0),-90}",
        "positionTo": "{(Breite/2),0,-42.5}"
      },...
],
    "ranges": [{
        "position": "{0,-(-34.5),1100}",
        "mask": "DockingMask1",
        "assignments": {},
        "rotation": "{0,0,0}",
        "stepEnd": "{0,34.5,2200}",
        "stepX": "0",
        "stepY": "0",
        "stepZ": "488"
      },...
],
    "lineRanges": [{
        "position": "{(-Breite/2),0,-42.5}",
        "mask": "DockingMask2",
```

```
        "assignments": {},
        "rotation": "{0,0,-90}",
        "positionTo": "{(Breite/2),0,-42.5}",
        "stepEnd": "{0,34.5,2200}",
        "stepX": "0",
        "stepY": "0",
        "stepZ": "488"
      },...
]
  },
  "childDockings": {
    "points": [...]
  },
  "addOnSpots": [{
      "position:"{-100,0,0}",
      "visible":"hasLeftNeighbour == false",
      "mask":"leftSpot"
    },...
  ],
  "subComponents": [ {
    "internalId":"leftSide",
  "componentId":"sampleCatalog:sideComponent",
  "assignments": {
    "Length":"Length"
  },
  "numberInPartList":"3",
  "active":"true",
  "supersedings":[
  { "type":"parameter","key":"color" }
  ]
  },{
    "internalId":"rightSide",
  "componentId":"sampleCatalog:sideComponent",
  "assignments": {
    "Length":"Length*2"
  },
  "numberInPartList":"if(Length > 10) {number= 2; } else { number = 1;
     }"
  },...
  ],
  "packageSizes": [5, 3],
   "packaging": [{
       "size": 1,
       "condition": "someOtherParameter == 1"
   },{
```

```
        "size": 7,
        "condition": "someOtherParameter == 1"
    }],
  "articleNr":"article1234",
  "geometry": "<geometry skript>",
  "geometryHD": "<geometry HD skript>",
  "previewGeometry": "<previewGeometry skript>"

}
```

tag name	type	description
id	string	the global unique id of the component. It is combined from the unique id of the catalog and an unique id of the component within the catalog
onUpdate	Script	a Script being executed on every changes. Setting of parametervalues within this script persist.
valid	Script	a condition to evaluate if the component is considered valid.
labels	map	a map containing key-value pairs for the label of this component. The keys are the isocodes of the language. The values are the labels to be used.
parameters	array	list of the possible parameters of this component
parameterGroups	array	list of the possible parameterGroups of this component.
possibleChildren	array	list of the possibleChild objects defining possible children for this component. Each child must have either itemId or componentId. Items must be configurable. Can have a script defining if it is the default. If multiple possible children evaluate default to true, its undefined which one is taken as the default child.
parentDockings	map	contains the definitions for all possible dockings where other components can be docked to this component as children. There can be points, lines, ranges (of points) and lineRanges.
childDockings	map	contains the definitions of all possible dockingPoints where this component can be docked to another component as a child. This contains only points.
addIbSoits	array	contains the definitions of all possible addOnSpots.
subComponents	array	list of possible subcomponents of this component. They can be referenced from the geometryScript and the previewGeometry by the internalId. If the component itself should appear in the partlist additionally to the subcomponents, a backreference is possible.
articleNr	string	the script to calculate the articleNr that should be displayed in the partlist. If the script contains only a string, this string is used as the articleNr. Otherwise the content of the variable "articleNr" is used.

packageSizes	array	list of numbers of packages which are allowed for this component
packaging	array	list of sizes with conditions for adding certain sizes if needed
dimensionings	array	list of all dimensionings that might be used.
siblings	array	contains the definitions of all possible siblingPoints.
geometry	string	the geometry script for this component. For details see the RoomleScript chapter.
geometryHD	string	the geometry script for higher quality of this component. For details see the RoomleScript chapter.
previewGeometry	string	the geometry script for this component in previewmode. For details see the RoomleScript chapter.

2.4 Measuring of Components

By default the Configurator takes the real bounding box of the geometry as the measurements. If necessary the scripter can provide a custom definition for the shown measurements with the setBoxForMeasurement command in the onUpdate script. When the command is called, the display of the measurements behaves as if the component would consist only of a simple Cube with the size and offset as given in the command.

2.5 Combining Components

Components can be docked to each other via dockpoints. Each Component can have at most one parent which leads to a parent-child treehierachy. It is possible to transfer data from one component to another. Either directly via the Dockingconnection or via Siblingconnections which can be created between any two components. Data is transfered via assignments.

2.5.1 Docking

Each component can define dockingPoints and dockingLines where other components can be docked to (called ParentDockings) and dockingPoints (called ChildDockings) through which this component can be docked to the ParentDockings of other components. Components which are docked to another component A are called the children of A. Through the docking connection the parent can set parameters of the child, e.g. the width of a shelf can be set by the parent frame. The child can also set parameters of the parent. There are three types of assignments: onDock, onUpdate and onUndock.

When a new component is docked the assignments are always first executed on the parent side. Meaning

- assignmentOnDock in parentDocking

- assignmentOnDock in childDocking
- assignmentOnUpdate in parentDocking
- assignmentOnUpdate in childDocking

Assignments in the onDock Block doesn't trigger recursive onUpdate actions themself.

On configuration changes, only the update assignments from the changed component are executed.

On Undock the order is reversed:

- assignmentOnUnDock in childDocking
- assignmentOnUnDock in parentDocking

One ParentDockPoint can only be used by one ChildDockPoint, on the other hand one ParentDockLine can be used by multiple ChildDockPoints at once. The maximum possible Children on a DockLine can be set via the maxChildren property.

2.5.2 Siblings

Additionaly to assignments throu parent/child connections, components can transfer Data via Siblingpoints. Siblingpoints connect to any other siblingpoint with the same mask and same position somewhere within the object. Therefor its possible to transfer data (parametervalues) directly between two childcomponents on completly seperated branches in the parent-child-hierachy. Like Dokcings, Siblingpoints have onDock, onUpdate and onUndock assignments. Regarding the executionorder during the docking process the logic follows the same rules as with Dockingpoints: first the assignments from the currently existing Component, then the assignment from the newly docked one. First both onDock, then onUpdate. On undock the order is reversed, same as with the DockingPoints.

Docking assigments are always executed prior to sibling assignments.

2.5.3 Self Assignments

Normal assignments (either on docking or siblings) sets the value of a parameter on the other side of the connection. With self assignments one can set values on parameters of the component of the docking/sibling.

Self assignments are always executed after the "normal" assignment and never trigger recursive actions.

2.5.4 Silent Assignments

Normal "On Update"-Assignments lead to a disabling of the parameter which is set from the assignment. This is done because normally a User shouldnt be able to change a Parameter whose value will be override with the next refresh. In some case (e.g. if the change of the value

triggers an update of the connected Component which leads to changes in the parameters so that the changed value stays the same after the next refresh) this automatic disabling should be silenced. In this case one can use assignmentsOnUpdateSilent. Those assignments are handled exactly like onUpdate Assignments, but without disabling the parameter. Silent assignments are always applied before the normal assignments (if both exist).

2.5.5 AssignmentScripts

For more complex solutions its possible to define assignmentScripts. Again seperated into onDock, onUnDock and onUpdate, those scripts are executed after the "normal" assignments are done. Within the assignmentScript one has access to the values of both sides of the connection (parent and child in docking, both siblings for siblingpoints). Those are accessilbe via context definition. F.e. the paramter "width" of the other side of the connection is accessible via "other.width" while "self.width" is the own parameter width. In parentDockings the "other." context is also accessible via "child.", in childDockings its "parent." and for Siblings its "sibling."

2.5.6 Dock Ranges

Dockranges provide the ability to create a range of dockPoints without specifing each of them individually. A range is defined by a startingPoint, the stepSize and the endPoint. The step might be one 3D step which is used as a direct increment until the endPoint is reached, or as stepX, stepY and stepZ which creates a raster of points. The condition and assignments are defined for the range, but executed for each generated point seperatly. Within those scripts you can access the position of the point and index within the range via "connection.position" and "connection.index".

2.5.7 AddOn Spots

For a visual hint to the user, its possible to define addOnSpots. An AddOnSpot is a position in the 3D space relative to the current Component where a Plus-Sign is shown. If the User clicks on the sign, the AddOn-View opens.

AddOnSpots have a position and a condition. The mask is defined for future use and has nothing to do with docking masks.

2.6 Packaging Size

This feature allows defining a package size for its component. This size will be the amount of all appearances of this component. Description by example:

```
{
    "id": "comp",
    "parameters": [
```

```
        {
            "key": "someParameter",
            "type": "Decimal",
            "visible":true,
            "defaultValue": 1,
            "validValues": [1, 2, 3, 4, 5, 6, 7, 9, 10]
          },
        {
            "key": "someOtherParameter",
            "type": "Decimal",
            "visible":true,
            "defaultValue": 0
         }
    ],
    "geometry": "
        ...
    ",
    "parentDockings": {
       ...
    },
    "packageSizes": [5, 3],
    "packaging": [{
        "size": 1,
        "condition": "someOtherParameter == 1"
    }],
    "articleNr": "
      articleNr= 'nr'|someParameter|'x'|packageSize
    ",
    "pricing": [{
        "region": "default",
        "currency": "EUR",
        "price": "
           price= 100*packageSize;
         "
    }]
}
```

The possible "packageSizes" are set to [5,3], this means for example if the component is docked 11 times the number of packages with size 5 will be 2 and the number of packages with size 3 will be 1. The number of components will always be first divided into the biggest possible package size and then the next smaller one and next smaller one until the smallest one is reached.

The parameter packaging adds also the size 1 to the "packageSize" if the condition of "someOtherParameter == 1" is true.

The parameter "articleNr" and "pricing" are containing the variable "packageSize". There-

fore it is possible to get the current “packageSize”.

2.7 Dimensioning

The configurator automatically shows the dimensions of the bounding box, or of the boxForMeasurement if defined. Additional dimensioning levels can be defined as objects in the dimensionings array. Every dimensioning has a type, from, to, level and visibility field:

- type defines where this dimensioning applies (x, y or z)
- from and to defines the begining and the end of the dimensioning
- level is the layer of the dimensioning. 0 is the outermost layer
- visible can be any condition defining if this dimensioning should be shown.

2.8 Configuration format

The Configuration describes an actual component in a plan with the set parameters and the docked children.

```
{
  "componentId": "<id der Root Komponente>",
  "parameters": {
        "paramKey1": "paramValue1",
        "paramKey2": "paramValue2", ?
  },
  "children": [{
         "componentId": "",
         "parameters": {
               "paramKey1": "paramValue1",
               "paramKey2": "paramValue2", ...
        },

// Docking point sample
        "dockParent": "[ {x,y,z} ]",
        "dockPosition": "{x,y,z}",
        "dockChild": "{x,y,z}",

// Docking line sample
        "dockParent": "[ {x,y,z}, {x,y,z} ]",
  "dockPosition": "{x,y,z}",
        "dockChild": "{x,y,z}",

        "children": [ ... ]
```

```
    }]
}
```

Tag name	Type	Description
componentId	string	the id of outermost component.
parameters	map	the values to be set for the parameters of this component. Its a Map of key-value pairs where the key is the key of the parameter and the value is the value to be set.
children	array	contains recursivly the definition for all components which are docked to this component.
children/dockChild	string	defines the dockingPoint (one of the points defined in ChildDockings) used to dock this component to its parent. It is defined throu the position of the dockingPoint.
children/dockParent	string	defines the docking of the parent (from the possible dockings defined in ParentDockings of the parent) used to dock this component to. For a DockingPoint the Array contains one point (position of the point), for a DockingLine it contains two points (the start and end point of the line). The positions are within the coordinate system of the parent.
children/dockPosition	string	defines the actual position of the dockingPoint within the coordinate system of the parent. This is used to define the actual position on a dockingLine.

A note to the docking logic: The positions should be stated with 2 digits floating point precision. When matching the points with given docking points from the component definition, they are rounded to 1 digit.

2.9 Requirements

There are a few requirements to consider when defining the configuration definition for Components. Since Roomle provides realtime visualization on different plattforms, the models must not become to complex. In general the same rules apply as for static items. When talking about 3D visualization the final number of triangles is the main factor. The less triangles the better the performance. On the other hand less triangles may mean less quality. Therefore one should thrive for the perfect tradeoff between quality and performance.

We define recommendations for preferred average values and upper boundaries that must not be exceeded.

2.9.1 Items

Since Components can be combined to items (via docking logic) of a big variety in size and complexity are possible. Nevertheless the average trianglecount should be between 2000 and 3000. Any Item must not exceed 10000 triangles.

For these values only the rendered triangles are counted, meaning the geometry parts send to the 3D engine for a given parameter combination. Meshes and Primitives which are in the geometryscript but not send to the Engine because of conditions in the script are not counted. Because of the nature of primitives, they should always be preferred to meshes wherever possible.

Geometryfunction	trianglecount desktop	trianglecount mobile
AddCube	43	12
AddSphere	Depending on the maximum size per dimension: $<$ 10 mm: up to 80 $<$ 100 mm: up to 290 $<$ 1000 mm: up to 1100 $<$ 3000 mm: up to 2100	same
AddCylinder	8*CircleSegments - 4	4*CircleSegments - 4
AddPrism	8*NumberOfVertices - 4	4*NumberOfVertices - 4
AddRectangle	2	2
AddMesh	as defined in the call if no indices provided: nrVertices/3	as defined in the call if no indices provided: nrVertices/3

2.9.2 Configuration Definition

Beside the restrictions on triangles for realtime rendering performance, restriction on filesize for the configuration format apply. This is needed to ensure parsing and execution speed during interaction. Configuration definition should have less than 10k characters on average and must not exceed 500k characters. For these values all characters in the json-file are counted. To fullfill those boundaries one should try to use primitives wherever possible.

2.9.3 Textures

Since imagefiles are always heavy on the memory usage, please keep the usage of textures as little as possible.

Regarding imagefiles for the textures, the same rules as for textures of static items applies:

Imagefiles must be JPEG or PNG with a maximum size of 2048x2048 pixels. The recommended size is below 512x512 pixels. For better performance consider keeping the imagesize

a power of two (e.g. 32,64,128...), not necessarily square and use the mm dimensions for scaling.

Chapter 3

JSON Objects

This chapter defines all possible JSON Objects used within a Component definition or Configuration. A "?" denotes an optional field.

```
Currency: "EUR"|"GBP"|"USD"|...
Region: "default"|"at"|"uk"|"us"...
Language: "en"|"de"|...
Type: "Integer"|"Decimal"|"String"|"Boolean"|"Material"
UnitType: "length"|"area"|"count"|"angle"
Value: long|float|String|true|false
Script<ResultType>: String

Range : {
  "valueFrom": long|float,
  "valueTo": long|float,
  "step": long|float
}

ParameterGroup : {
  "key": String,
  "labels": { (Language(_Country)?: String)+ },
  "collapsed": Script<Boolean>?(false),
  "sort": Script<Long>?(0)
}

Parameter : {
  "key": String,
  "type": Type,
  "sort": long,
  "unitType": UnitType,
  "value": Value,
  "labels": { (Language(_Country)?: String)+ },
  "defaultValue": Value?,
  "enabled": Script<Boolean>?(true),
```

```
  "visible": Script<Boolean>?(true),
  "visibleAsGlobal": Script<Boolean>?(true),
  "global": Boolean?(false),
  "highlighted": Boolean?(false),
  "visibleInPartList": Script<Boolean>?,
  "validGroups": [ String* ]?,
  "validRange": Range?,
  "validValues": [ Value* ]?,
  "conditionalGroups":[ ValueObject* ]?,
  "valueObjects":[ ValueObject* ]?,
  "onValueChange":Script?,
  "group":Script<String>?
}

ValueObject  {
 "value": Value,
  "labels": { (Language(_Country)?: String)+ }?,
  "condition": Script<Boolean>?,
  "thumbnail":String?
}

ConnectionWithAssignment  {
 "position": Script<Vector3f>,
  "mask": String,
  "assignmentsOnDock": { String:Script< String>,...},
  "assignmentsOnUpdate": { String:Script< String>,...},
  "assignmentsOnUpdateSilent": { String:Script< String>,...},
  "assignmentsOnUnDock": { String:Script< String>,...},
  "selfAssignments": {
    "onDock": { String:Script< String>,...},
    "onUpdate": { String:Script< String>,...},
    "onUnDock": { String:Script< String>,...}
  },
  "assignmentScripts": {
    "onDock": Script,
    "onUpdate": Script,
    "onUnDock": Script
  }
}

DockingPoint : ConnectionWithAssignment {
  "rotation": Script<Vector3f>,
  "condition": Script<Boolean>?(true)
}

DockingRange  {
```

```
 "stepX": Script<float>,
  "stepY": Script<float>,
  "stepZ": Script<float>,
  "stepEnd": Script<Vector3f>
}

DockingPointRange : DockingPoint, DockingRange  {
}

DockingLine : DockingPoint {
  "positionTo": Script<Vector3f>,
  "maxChildren":Script<long>?(Inf)
}

DockingLineRange : DockingLine, DockingRange {
}

ParentDockings : {
  "points":[ DockingPoint* ],
  "lines": [ DockingLine* ],
  "ranges":[ DockingPointRange* ],
  "lineRanges": [ DockingLineRange* ]
}

ChildDockings : {
  "points":[ DockingPoint* ]
}

SiblingPoint : ConnectionWithAssignment {
}

AddOnSpot {
  "position": Script<Vector3f>,
  "mask": String,
  "condition": Script<Boolean>?(true)
}

PriceList : {
  "region": Region,
  "currency": Currency,
  "retailPriceDependsOnCustomerPrice": Boolean?(false),
  "price": Script,
  "retailPrice": Script
}
```

```
Superseding : {
  "type": String,
  "key": String
}

SubComponent : {
  "internalId": String,
  "componentId": String,
  "assignments": { String:Script< String>, ...},
  "numberInPartList":Script<integer>,
  "active":Script<Boolean>?(false),
  "supersedings":[ Superseding* ]
}

PossibleChild : {
  "itemId": String?,
  "componentId": String?,
  "condition": Script<Boolean>?(true),
  "default": Script<Boolean>?(false)
}

Component : {
  "id": String,
  "valid": Script<Boolean>?(true),
  "labels": { (Language(_Country)?: String)+ },
  "parameters": [ Parameter* ],
  "parameterGroups": [ ParameterGroup* ]?,
  "possibleChildren": [PossibleChild*],
  "parentDockings": ParentDockings?,
  "childDockings": ChildDockings?,
  "siblings":[ SiblingPoint* ]?,
  "addOnSpots":[ AddOnSpot* ]?,
  "geometry": Script,
  "geometryHD": Script?,
  "previewGeometry": Script,
  "packageSizes": Array<Integer>,
  "packaging": [Packaging+],
  "dimensioning": [ Dimensioning*]?,
  "pricing": [ PriceList+ ],
  "articleNr":Script<String>,
  "subComponents":[SubComponent*],
  "onUpdate":Script?
}

ParameterValues  {
  String: String, ...
}
```

```
Configuration : {
  "componentId": String,
  "parameters": ParameterValues,
  "dockParent": Script<Vector3fArray>,
  "dockPosition": Script<Vector3f>,
  "dockChild":Script<Vector3f>,
  "children": [Configuration*]?
}

Packaging : {
  "size": Integer,
  "condition": Script<Boolean>
}

Dimensioning : {
  "labels": { (Language(_Country)?: String)+ },
  "type": String,
  "from": Script<Float>,
  "to": Script<Float>,
  "level": Integer,
  "visible": Script<Boolean>
}
```

Chapter 4

RoomleScript

RoomleScript is a simple scripting untyped language used within the component definition and configurations. There exists the possiblity of Comments either single line or multiline. Comments are ignored by the Configurator.

4.1 Grammar

```
Script: Command*

Command: Evaluate | Assign | IfElse | For | Comment | MultilineComment
  Evaluate: Expression ';'
  Assign: identifier '=' Expression ';'
  IfElse: 'if' '(' Condition ')' '{' Command* '}' ('else' '{' Command*
     '}')?
  For: 'for' '(' Command ';' Condition ';' Command ')' '{' Command* '}'
  Comment: ( '#' | '//' ) String* '\n'
  MultilineComment: '/*' String* '*/'

Condition:
  Condition ('&&' | '||') Condition
  Expression ( '==' | '!=' ) Expression
  Expression ( '<=' | '>=' | '<' | '>' ) Expression

Expression: SimpleExpression | Array
  Array: identifier? '[' (SimpleExpression(,SimpleExpression)*)? ']'

SimpleExpression: Variable | Function | Struct | Value | UnaryOperation
    | BinaryOperation | '(' Expression ')'
  Variable: String
  Function: FunctionName '(' Expression* ')'
  FunctionName: String
```

```
Struct: Vector2f | Vector3f
Vector2f: 'Vector2f'? '{' SimpleExpression ',' SimpleExpression '}'
Vector3f: 'Vector3f'? '{' SimpleExpression ',' SimpleExpression ','
   SimpleExpression '}'
Value: Number | String
Number: int | float
UnaryOperation: ( int '++' ) | ( ('sin'|'cos'|'tan') '(' Expression '
   )' )
BinaryOperation: SimpleExpression Operand SimpleExpression
Operand: '+' | '-' | '*' | '/'
```

Listing 4.1: Script Grammar

4.2 Parameters, internal Variables and Context

Within a component every script has access to a list of variables. All scripts can read the values but only onUpdate, onValueChange and the assignmentScripts are allowed to change them. The list of variables contain all Parameters with the parameter-key as the variable name. New variables are created on first use (e.g. when a value is assigned to it). In Scripts with write access to the internal variables those variables are also stored and from now on accessible from all the other scripts.

In some scripts you have access to more than one list of variables at once (for example in assignmentScripts). For that you can access variables via "<context>.<variableName>" . In DockRanges you can for example access the position of the current dockPoint via "connection.position" and the index within the dockRange via "connection.index".

4.3 Available Functions

This section includes all functions available in the Roomle Script

4.3.1 General Functions

Functionname	Description
getPosition	returns the current position of the component in the coordinate system of the parent
getDockPosition	returns the current position of the childDockPoint of this component in the coordinate system of the parent
getDockPositionRelativeToParentDock	returns the current position of the childDockPoint of this component relative to the parentDock in the coordinate system of the parent. The reference is the point for dockpoints and the starting point for docklines.
getUniqueRuntimeId	returns a unique runtimeId for the component. This id is unique in the session and wont change for this component (during this session)

Functionname	Parameters	Description
xFromVector	vector:Vector3f{x,y,z}	returns the x component of the given vector
yFromVector	vector:Vector3f{x,y,z}	returns the y component of the given vector
zFromVector	vector:Vector3f{x,y,z}	returns the z component of the given vector
set	array:Array<float> index:int value:float	sets the value at the given index in the array
get	array:Array<float> index:int	returns the value at the given index in the array
pushBack	array:Array<float> value:float	adds the value at the end of the array
popBack	array:Array<float>	removes and returns the last element of the array
indexOf	value:float array:Array<float>	returns the first index of the value in the array, or -1 if not found
length	array:Array<float>	returns the current length of the array

Functionname	Parameters	Description
setBoxForMeasurement	Box:Vector3f{w,d,h}, Offset:Vector3f{x,y,z}	defines the box to be used for calculating the measurements of this component
isEnabled	parameterKey:string	returns true if the parameter is enabled
setEnabled	parameterKey:string value:bool	sets the enabled-property of the parameter to the given value
isVisible	parameterKey:string	returns true if the parameter is visible
setVisible	parameterKey:string value:bool	sets the visible-property of the parameter to the given value
in	anyVariable listOfValues...	returns true if the first Parameter is equal to any of the remaining parameters
inArray	anyVariable array:Array<float>	returns true if the first Parameter is equal to any of the elements in the array
string	any	converts the parameter to string
float	any	converts the parameter to float
size	param:string	calculates the size of the given string
substring	string:string startIndex:int length:int	returns the substring of the given string, starting at startIndex with given length

Functionname	Parameters	Description
tanh	angle:float	returns the tanges hyperbolicus
sinh	angle:float	returns the sinus hyperbolicus
cosh	angle:float	returns the cosinus hyperbolicus
tan	angle:float	returns the tanges
sin	angle:float	returns the sinus
cos	angle:float	returns the cosinus
atan	value:float	returns the arcus tanges
atan2	y:float x:float	returns the principal value of the arc tangent of y/x
asin	value:float	returns the arcus sinus
acos	value:float	returns the arcus cosinus
log	value:float	returns the natural logarithm
log10	value:float	returns the logarithm with base 10
exp	value:float	returns the exponential function
fabs	value:float	returns the absolute value
sqrt	value:float	returns the square root
fmod	a:float b:float	returns the floating point remainder of a/b
pow	a:float b:float	calculates a to the power of b
floor	number:float, digits:int	returns the next lower number with the given digits
ceil	number:float, digits:int	returns the next bigger number with the given digits
round	number:float, digits:int	rounds to the nearest number with the given digits
isnull	anyVariable	returns true if the variable/parameter is not set
ifnull	anyVariable defaultValue:any	returns the first parameter if it is not null, otherwise the second

4.3.2 Geometry Object Functions

There exist a set of functions for generating geometry objects for this component. Each function has optional Parameters to modify the generation of the uv-Coordinates for the object. This is relevant for the placement of textures. There must either be ALL uv-modifications provided or none.

A uvScale of 2 means that the uv-Coordinates are scaled by 2 which results in the texture being shown with half the size. Same goes for the uv-Offset and Rotation: They modifiy the position and orientation of the uv-Coordinates, which in return affects the placement of the texture in the "other" direction.

For the AddMesh function one can provide explicit uvCoord and normals. The number of uvCoords must match the number of indices provided. The number of normals must match the number of vertices provided.

Vertices, uvCoords and uvOffset ist always in mm.

Parameters marked with * are optional.

Functionname	Parameters	Description
AddCube	Size:Vector3f{width,depth,height}, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a Cube
AddPlainCube	Size:Vector3f{width,depth,height}, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a sharp Cube without bevel
AddSphere	Size:Vector3f{width,depth,height}, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a Sphere
AddCylinder	RadiusBottom:float, RadiusTop:float, Height:float, CircleSegments:float, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a Cylinder
AddPrism	Height:float, Vertices:Array<Vector2f>, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates an extruded object given the 2D Shape and the height
AddRectangle	Size:Vector2f{width,height}, *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a flat rectangle
Copy		creates a copy of the last drawn object or group

Functionname	Parameters	Description
AddMesh	Vertices:Array<Vector3f>, *Indices:Array<int> , *uvScale:Vector2f{scaleU, scaleV } , *uvRotation:float , *uvOffset:Vector2f{offU, offV }	Creates a mesh from the given vertices. If no indices are provided, the number off points must be a multiple of 3. Every group of 3 vertices form a triangle. With indices there must not be an index bigger than the number of Points -1. The number of Indices must be a multiple of 3
AddMesh	Vertices:Array<Vector3f>, *Indices:Array<int> , *uvCoordinates:Array<Vector2f> , *normals:Array<Vector3f>	Creates a mesh from the given vertices with uvCoordinates and normals
AddExternalMesh	meshId:String Bounds:Vector3f{w,d,h}, BoundsOffset:Vector3f{x,y,z}	Adds an external Mesh with the given Id. Bounds and boundsOffset tells the configurator the size and position of the loading box until the mesh is loaded.
BeginObjGroup	GroupName:String	Begins a new objectgroup
EndObjGroup	GroupName:String	Defines the end of the Objectgroup
MoveMatrixBy	Translation:Vector3f{x,y,z}	Moves the previously drawn object or group by the given vector
RotateMatrixBy	RotationAxis:Vector3f{x,y,z}, Pivot:Vector3f{x,y,z}, Angle:float	Rotates the last drawn object or group according to the parameter
SetObjSurface	MaterialId:String	Assigns the given Material to the last drawn object or group
SubComponent	internalId:String	Injects the geometry of the given Subcomponent

4.4 Constants

Additionally you can use the usual math constants: M_E, M_LOG2E, M_LOG10E, M_LN2, M_LN10, M_PI, M_PI_2, M_PI_4, M_1_PI, M_2_PI, M_2_SQRTPI, M_SQRT2, M_SQRT1_2

Chapter 5

Examples

5.1 Component Definition of a table

This geometry script defines a table where the material of the plate and the legs can be set (via Parameter of the component). The width of the table is also a parameter.

```
Height= 750;
Width= 800;
TableThickness= 28;
LegHeight=682;
LegBuffer=16;
LegBufferGap= 5;
BeginObjGroup(group_table);
BeginObjGroup(group_legs);
  BeginObjGroup(group_leg);
    Cylinder(30,30,LegHeight-LegBuffer-LegBufferGap,20,1,1);
      MoveMatrixBy(Vector3f{0,0,LegBufferGap+LegBuffer});
    Cylinder(30,30,LegBuffer,20,1,1);
  EndObjGroup();
    SetObjSurface(LegMaterial);
  Copy();
    MoveMatrixBy(Vector3f{0,572,0});
 EndObjGroup();
  SetObjSurface(LegMaterial);
  MoveMatrixBy(Vector3f{114,Width/2-286,0});
Copy();
  RotateMatrixBy(Vector3f{0,0,1},Vector3f{0,400,0},180);
  MoveMatrixBy(Vector3f{Length,0,0});
Cube(Vector3f{Length-400,40,Height-TableThickness-LegHeight});
  MoveMatrixBy(Vector3f {200,Width/2-150,LegHeight } );
  SetObjSurface(LegMaterial);
Copy();
  MoveMatrixBy(Vector3f {0,300,0} );
```

```
Cube(Vector3f{Length-TableThickness*2,Width-TableThickness*2,
    TableThickness});
  MoveMatrixBy(Vector3f{TableThickness,TableThickness,Height-
      TableThickness+1});
  SetObjSurface(TableSurface);

Mesh(AlternatePlateAsMesh,Vector3f[
{0,0,TableThickness},{Length,0,TableThickness}, {Length,Width,
    TableThickness},
{0,0,TableThickness},{Length,Width,TableThickness}, {0,Width,
    TableThickness},
{0,0,TableThickness}, {TableThickness,TableThickness,0}, {Length,0,
    TableThickness},
{TableThickness,TableThickness,0}, {Length-TableThickness,
    TableThickness,0}, {Length,0,TableThickness},
{Length,0,TableThickness},{Length-TableThickness,TableThickness,0}, {
    Length, Width,TableThickness},
{Length-TableThickness,TableThickness,0}, {Length-TableThickness,Width-
    TableThickness,0},{Length,Width,TableThickness},
{Length,Width,TableThickness},{Length-TableThickness,Width-
    TableThickness,0},{0,Width,TableThickness},
{Length-TableThickness,Width-TableThickness,0},{TableThickness,Width-
    TableThickness,0},{0,Width,TableThickness},
{0,Width,TableThickness},{TableThickness,Width-TableThickness,0},{0,0,
    TableThickness},
{TableThickness,Width-TableThickness,0},{TableThickness,TableThickness,
    0},{0,0,TableThickness},
{TableThickness,TableThickness,0},{Length-TableThickness,Width-
    TableThickness,0},{Length-TableThickness,TableThickness,0},
{TableThickness,TableThickness,0},{TableThickness,Width-TableThickness,
    0},{Length-TableThickness,Width-TableThickness,0}]);
  MoveMatrixBy(Vector3f {0,0, Height-TableThickness } );
  SetObjSurface(TableSurface);
EndObjGroup();
  MoveMatrixBy(Vector3f{-Length/2,-400,0});
```

5.2 Price calculation

After running the price calculation script provided in the component definition, the value of the variable "price" is used as the calculated price.

```
if(TableSurface == sampleCatalog:Material1 || TableSurface ==
    sampleCatalog:Material2)  {
        if(Length <= 1400){ price=123.00;}
```

```
        if(Length > 1400 && Length <= 1600) { price= 234.00;}
        if(Length > 1600 && Length <= 1800) { price= 456.00;}
        if(Length > 1800) { price= 789.00;}
} else {
        if(Length <= 1400){ price=245.00;}
        if(Length > 1400 && Length <= 1600) { price= 356.00;}
        if(Length > 1600 && Length <= 1800) { price= 467.00;}
        if(Length > 1800) { price= 589.00;}
}
```