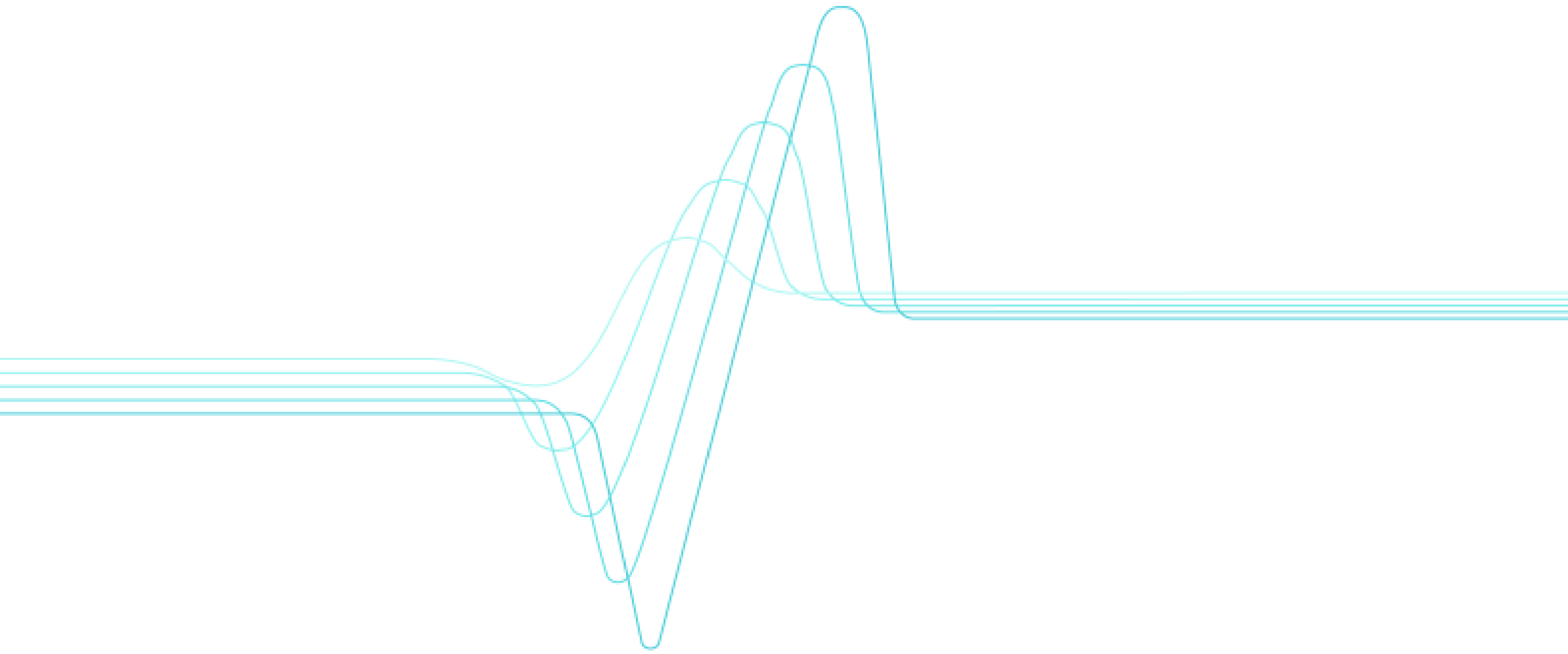
ESPOO 2003 VTT RESEARCH NOTES 2184

Kalle Kondelin & Tommi Karhela

Gallery Markup and Query Language Specification



VTT TIEDOTTEITA – RESEARCH NOTES 2184

Gallery Markup and Query Language Specification

Kalle Kondelin & Tommi Karhela

VTT Industrial Systems

ISBN 951–38–6117–1 (soft back ed.)
ISSN 1235–0605 (soft back ed.)

ISBN 951–38–6118–X (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Tuotteet ja tuotanto, Tekniikantie 12, PL 1301, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 6752

VTT Industriella System, Teknikvägen 12, PB 1301, 02044 VTT
tel. växel (09) 4561, fax (09) 456 6752

VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 6752

Technical editing Maini Manninen

Otamedia Oy, Espoo 2003

Kondelin, Kalle & Karhela, Tommi. Gallery Markup and Query Language Specification. Espoo 2003. VTT Tiedotteita – Research Notes 2184. 111 p.

Keywords Gallery Markup Language, Gallery Query Language, software infrastructure

Abstract

Gallery is a software infrastructure for storing, deploying and using process model and parameter data. This document is compiled for Gallery model server and client developers as well as for Gallery client and server extension developers. It describes the data model (GML) and main interfaces (GQL) of the Gallery system. Gallery was developed in a research project (2000-2002) funded by National Technology Agency (TEKES) and Finnish industry. This document is one of the end products of the project.

Contents

1. Introduction

Gallery is a software infrastucture for storing, deploying and using process model and parameter data. This document is compiled for Gallery model server and client developers as well as for Gallery client and server extension developers. This document describes the data model (GML) and main interfaces (GQL) of the Gallery system. The system architecture is described in (Karhela 2002) and it is considered as required knowledge for this document.

Gallery was developed in a research project (2000-2002) funded by National Technology Agency (TEKES) and Finnish industry. This document is one of the end products of the project. The data model and interfaces will be developed further in a future research project at VTT.

The basic components of the architecture are model servers and model clients. Model servers provide model configuration, simulation and database services for the model clients. The model clients provide user interfaces to these services. The interface to the model server services is a functional interface encapsulated into XML requests and responses. These XML requests and responses form the Gallery Query Language, GQL, described by this document. These functions are used for a wide range of services (adding categories, adding components, getting components, and so on).

To better understand the functions, this document also describes Gallery Markup Language, GML. GML describes the Gallery data model that the model servers control. The Gallery data model is only a logical data model, the physical data model may or may not correspond to it. No DTD or XML Schema has been been defined for it, although DTDs for some parts of the data model have been defined (e.g. for component types and components).

A central part of the Gallery system is the Gallery web site, which uses http protocol (RFC 2616) for transferring data to/from clients (Gallery 2002). There are few ASP pages (/gallery/bin/MSM.asp, /gallery/bin/GQL.asp) among the normal HTML and ASP pages hosted in the Gallery web server that provide access to the Gallery model servers hosted on the web site. This document does not describe the Gallery web site or what it is for. However, since this document describes what kind of data moves between the Gallery model servers hosted on the Gallery web site and the Gallery model clients, a few words about Gallery are in order to understand the context of this document. For proper and correct description of Gallery and what it is all about see (Gallery 2002).

In short, Gallery is a web site for accumulating and maintaining process component and model data. The components could be for example different kinds of pumps, valves, heat exchangers and so on. The primary users of the Gallery web site are component providers and model configurators. Component providers store data about their components into Gallery. Model configurators use this data by designing process models that contain components from Gallery using Gallery-aware applications. Gallery is specifically designed to support the use of the process component data in process simulation applications.

Gallery has a few typical use case scenarios:

- The kernel developer adds departments, categories, client and server extensions, and common definitions to Gallery. He/she first adds a department and then defines the structure of the department in question by adding categories. She also adds users and puts them into groups. Groups are the means of controlling access to the data in the department, and by defining groups the kernel developer can define a security policy for the department in question. The groups are then used by providers and model configurators when controlling access to the data they add to Gallery.
- The provider, working within the framework set by the existing categories and component types adds new component types. She can also add components or instances of component type. The instances can be equiment data or template models.
- The model configurator configures a model searching, fetching and using component and template model data from Gallery.
- The model user loads configured models and uses them i.e. simulates the models, observing the behavior of the model. A model configurator or user does not have to be a human user, a simulation application can as well fetch parameter values for components or template models from Gallery and use them.

and a few typical clients:

- Web browser (Internet Explorer, Netscape).
- Gallery’s own model client (Model Explorer).
- Simulation applications and their user interfaces (Apros).

Logically, there is a huge XML document that contains all the component data behind a model server. The GQL requests operate on this document. Therefore, the structure of this document (GML) is described first. Then the languge (GQL) used to operate on this data model is described.

2. Gallery Data Model

2.1 Introduction

This chapter explains the Gallery data model. The model described here is a logical model of the data in Gallery. Physical representation of this data is an implementation detail and it is not considered here. It can be an XML file, a relational database or anything else you can think of, all that matters is that the Gallery model server will honour the Gallery query language (GQL). The description of the logical data model is still necessary to both Gallery server developers and Gallery user interface developers, since the query language operates on this logical data model. The data model is represented in Figure 1.

The XML element and attribute names in the document are sometimes written using long names, but in the actual XML that moves between clients and servers the names are always short. The short element and attribute names are marked after the elements and attributes with brackets (like this: Administration [A]). The attribute values in the XML document always start with a lower case letter when the value represents a limited value set, even if it is written with upper case in this document for style reasons.

2.2 Security and Roles

The security is based on protecting the data moving in the Internet with SSL/TLS (Secure Socket Layer / Transport Layer Security). The client is authenticated with user identifier and password. The user identifiers are the operating system (W2K) user identifiers and they are used in deciding whether the client is entitled to use the Gallery services or not. The Gallery administrator must create the active user identifiers to the operating system. However, it is not enough that you know a user id and password, to be able to use the service requests you must also have one of the following roles:

- KernelDeveloper: Manages the creation of departments and users.
- Provider: Can add, remove and update client and server extensions
- ModelConfigurator: Can add, remove and update component types, enumerations, categories, components, and use extensions for manipulating data.
- ModelUser: Can only read and browse data, and use extensions for searching and reading data.

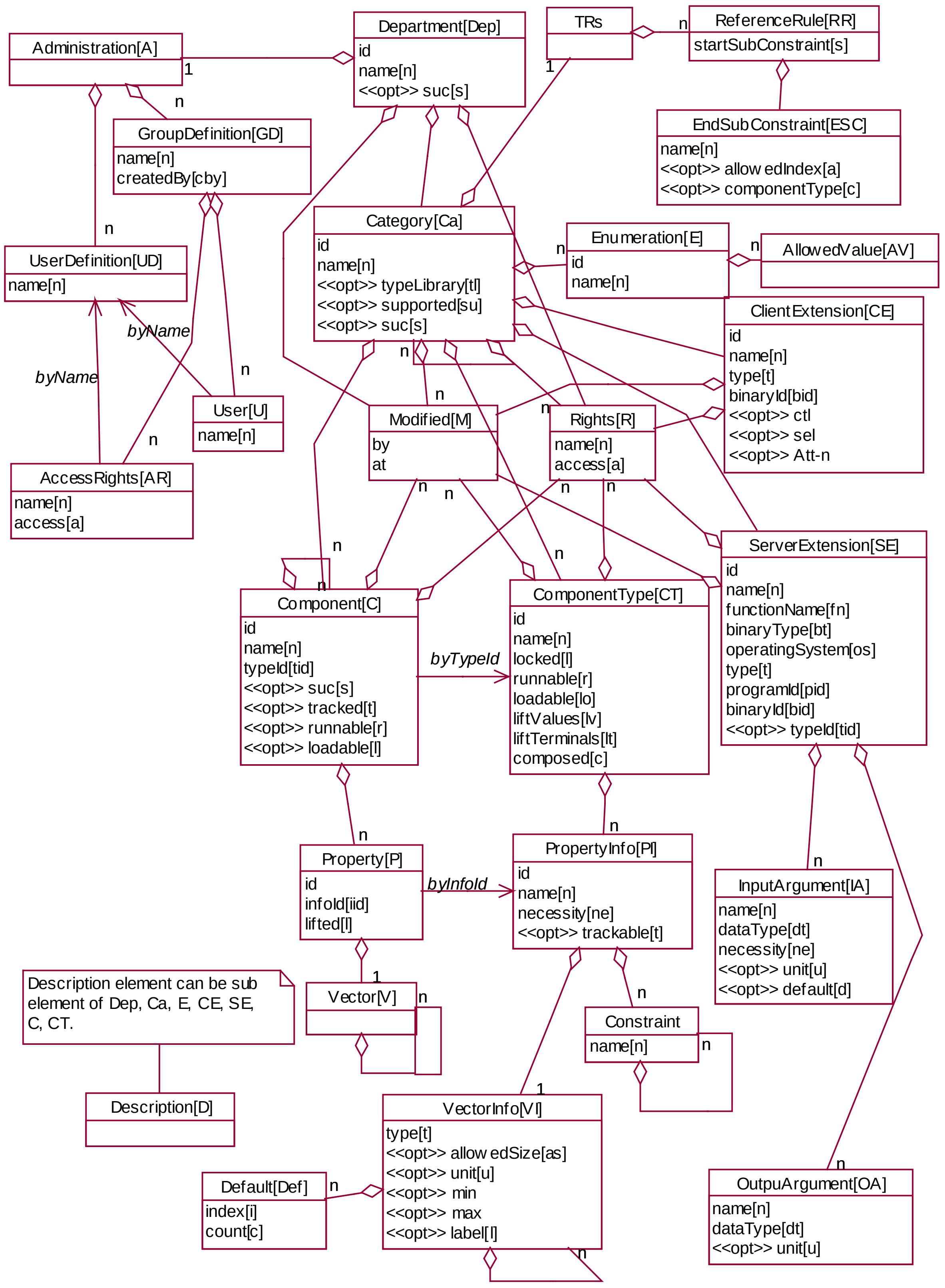


Figure 1. Gallery data model (GML).

After the administrator has inserted a user into one of the roles, the user can start to use the services with the rights that the role in question has. The role rights are nested so that if you belong to one role you automatically have the same rights as the inferior roles and a bit more. The role rights are fixed and hard coded into the server. In addition to this fixed server based security model, GML contains its own security model that will be described in the next paragraph.

2.3 Gallery Data Model Security

The backbone of the Gallery security model is the administration element and its user definition and group definition child elements. Only the administrators can create users by creating user accounts and assigning them to roles. After the physical accounts have been created, the kernel developers can add corresponding user definition elements to the desired departments. For security reasons the creation of user accounts and the assignment of roles can not be done through GQL. The user account name and the name of the user definition element must be same.

After the kernel developer has added the user definition elements the model configurators can then add and delete group definition elements. The group definition elements define groups and their users. The rights elements use groups when defining the rights for common elements like categories, component types and components.

The users of the system (UD elements) are unique in the system and groups (GD elements) are unique in the department in question. A user can be added to a group only once. The user can see/access only those departments she is a member of and has been given at least read rights. The users are grouped into groups. In each department there is a group named “Everyone” that contains all users of that department. The creator of the group has full rights to manipulate it and can give rights to others by adding access rights (AR) elements. Each AR element (that corresponds to a user) can be added only once to the group, just like the user element.

The rights of the Department, Category, ClientExtension, ServerExtension, ComponentType, Enumeration, and Component elements are given by adding the Rights element as their child element. The creator of an element has full rights to its manipulation. The creator gives rights to others by defining the Rights child element to the element in question. The Rights element defines the group in which the users can use the given rights. Each group can be added only once under a given parent element.

2.4 Element Descriptions

2.4.1 Introduction

The following element descriptions are short, reference like descriptions of the elements in the data model. Each description contains a list of attributes, a list of immediate child elements for the element in question and a short description. An attribute id is common in many elements. It uniquely identifies an element. The client is responsible for creating all active ids . The ids must conform to the UUID concept. UUID is a 16-byte value (36 bytes in character format) using hexadecimal digits in the following format: `12345678-1234-1234-1234-123456789ABC`. The client must prefix the UUID with “a” so that the id conforms to the XML id- specification. The RPC run-time libraries use UUIDs to check for compatibility between clients and servers and to select among the multiple implementations of an interface. When the id is used as a reference, empty string is used to express a reference to nothing.

In the following descriptions the order of the child elements is free, i.e. it does not matter what the description order of the child elements is, the elements returned by a model server can be in any order. The same applies to the attribute order.

2.4.2 Administration elements

2.4.2.1 Administration [A]

Child elements: UserDefinition, GroupDefinition.
Description:
As the name suggests, this element contains elements mainly for administrative purposes.

Example:
```
<Administration>
  <UserDefinition name=”Kernel01”>Kernel Developer.</UserDefinition>
  …
  <GroupDefinition name=”ToughGuys” createdBy=”Kernel01”>
    <AccessRights name=”Provider01” access=”write”/>
    …
    <User name=”Kernel01”/>
    …
  </GroupDefinition>
</Administration>
```

2.4.2.2 UserDefinition [UD]

Attributes:
- Name [n]: Name of the user. Must be an existing user account in the system. (The account must be assigned to some role.)

Description:
UD defines the users of the department.

2.4.2.3 GroupDefinition [GD]

Attributes:
- Name [n]: Name of the group.

- CreatedBy [cby]: Tells who has created the element.

Child elements: AccessRights, User.
Description:
GD defines groups of the department. (Rights elements use groups.)

2.4.2.4 AccessRights [AR]

Attributes:
- Name [n]: Name of the user. Refers to the corresponding name in the user definition element.

- Access [a]: The value can be “read” or “write”. Read means you can see but not modify the parent group definition. Write means you can read and modify the parent group definition.

Description:
When a group is created only the owner has rights to modify it. By adding AR elements the owner can give rights for others to modify the group definitions.

2.4.2.5 User [U]

Attributes:
- Name [n]: Name of the user. Refers to the corresponding name in the user definition element.

Description:
U defines the users belonging to the parent element.

2.4.3 Department elements

2.4.3.1 Department [Dep]

Attributes:

- Name [n]: Name of the department.

- Suc [s] (optional): The value can be “true” or “false”. If exists and the value is true then the department is structural. The department is structural if it has Category elements. This attribute is mainly for the GetTree function that does not necessarily return the category elements.

- Id [id]: UUID of the element.

Child elements: Description (optional), Administration (only one), Rights, and Modified.
Description:
The logic behind the Department element is that it will group data for a group of people working for a common object. The kernel developers and providers design a framework for the model configurators and users by defining common categories and component types. The providers can then insert equipment data to the categories, the model configurators can configure models, and the model users can start using the models.

Example:

```
<Department name=”My department” id=”UUID”>
    <Modified/>
    …
    <Rights/>
    …
    <Administration>…</Admininstration>
    <Category>…</Category>
    …
</Department>
…
```

2.4.3.2 Description [D]

Content: A descriptive text.

Description:
D defines the semantics of the parent element.

Example:

```
<Description>Once upon a time…</Description>
```

2.4.3.3 Category [Ca]

Attributes:

- Id [id]: UUID of the element.

- Name [n]: Name of the category. Must be unique within the containing element.

- TypeLibrary [tl] (Optional): If exists then this is a type library. Each department can only have exactly one type library and the type library must be the immediate child of the department. The type library contains the client extensions, server extensions and component types of the department. Also hierarchical components can be a part of the type library.

- Suc [s] (optional): Exists if the category is structural. The category is structural if it has other elements than Description, Modified, Rights or TerminalRules. This attribute is mainly for the GetTree function that does not necessarily return the sub-category elements.

- Supported [su] (Optional). If exists and true then the component types and their instances (structural component types) are supported (i.e. can be simulated) by the model server. Only the type library category can have the supported attribute.

Child elements: Category (optional), ComponentType (optional), TerminalRules (optional), Description (optional), Enumeration (optional), Rights (optional), ClientExtension (optional), ServerExtension (optional). The TerminalRules element can only exist in categories that have the type library attribute.

Description:

The kernel developers use this element to divide the component types into logical groups. The modified element is added automatically by the server. It keeps track of modifications, i.e. changes of attributes, adding and removing child elements. The first modified element is the creator, i.e. the owner of the category.

Example:

```
<Category name="Pumps" id="…">
  <Modified …>Created.</Modified>
  …
  <Rights …>
  …
  <Description>The description of pumps category.</Description>
  <Category …>
  …
  </Category>
</Category>
```

2.4.3.4 Rights [R]

Attributes:

- Name [n]: Name of the group. Refers to the corresponding name in the Administration / Groups / GroupDefinition element.

- Access [a]: Access rights for the containing element. The value can be “read” or “write”. Read means you can see (read) the elements and write means you can see and change the elements.

Description:

R defines the access rights for the containing element. Only elements with owner (Modified child element) can have a rights element. The owner of the containing element has full rights to the element, others have the rights defined by the rights elements (this includes the right to access the rights elements themselves). The allowed parents are Category, Enumeration, ClientExtension, ServerExtension, ComponentType and Component.

Example:

```
<Category …>
    <Rights name=”Everyone” access=”read”/>
    <Rights name=”My Group” access=”write”/>
</Category>
```

2.4.3.5 ComponentType [CT]

Attributes:

- Id [id]: UUID of the element.

- Name [n]: Name of the component type.

- Locked [l]: The value can be true or false. True means that only the data of the component type can be changed. False means that instances of this component type cannot be made, and that the data of the component type cannot be changed.

- Runnable [r]: If true, then the components of this type can be simulated. Only runnable components support Simulate/Freeze functions.

- Loadable [lo]: If true, then the components of this type can be loaded. Only loadable components support the Load, UnLoad and Save functions.

- LiftValues [lv]: If true, then value properties and mapping properties can be lifted from a lower level.

- LiftTerminals [lt]: If true, then start and end terminals can be lifted from a lower level.

- Composed [c]: If true, then subcomponents can be added.

Child elements: PropertyInfo, Modified (at least one), Rights (optional), Description (optional).
Description:
CT describes the type of the components. The description element is used to describe the semantics of the used property infos (and their vector infos and constraints) unless is already described by this document, or the category, component type or component descriptions. The modified element is added automatically by the server. It keeps track of modifications, i.e. changes of attributes, adding and removing of child elements. The first modified element is the creator, i.e. owner.

Release Notes For Gallery Setup 1.0:

LiftValues, LiftTerminals, and composed attributes are not implemented. Also Simulate, Freeze, Load, Save and Unload functionalities mentioned above are not implemented.

Example:

```
<ComponentType id…>
  <Modified …/>
  …
  <Rights…/>
  …
  <Description>…</Description>
  <PropertyInfo id=”UUID” name="PUMP_ROTATION_SPEED" …>
  …
  </PropertyInfo>
  …
</ComponentType>
```

2.4.3.6 Modified [M]

Attributes:
- At [at]: When the modification was made. Added automatically by the server.

- By [by]: Who made the modification. Added automatically by the server.

Content: A descriptive text of the modification.
Description:
The client can keep a change log with Modified elements. See description of component type for more information. The allowed parent elements are the same as for the Rights element.

Example:
```
<Modified at=”…” by=”…”>Created.</Modified>
```

2.4.3.7 PropertyInfo [PI]

Attributes:

- Id [id]: UUID of the element.

- Name [n]: Name of the property info.

- Necessity [ne]: Identifies the behaviour of the corresponding property in instantiation (set component function). The value can be “required”, “implied” or “optional”. Required means that the client must give it at instantiation. Implied means that the server creates a default instance (generating an id for the instance. Optional means that the client does not have to give it in instantiation process and the corresponding property is not created.

- Trackable [t] (optional): If exists and true, then operations that change the value of the corresponding property can cause a creation of a new Modified child element to the Component (if it is in the tracked mode).

Child elements: VectorInfo (only one, compulsory), Constraint.
Description:
PI describes the common features of properties.

Release Notes For Gallery Setup 1.0:
Trackable attribute is not implemented.

Example:
```
<PropertyInfo id=”UUID” name=”…” necessity=”implied”>
  <VectorInfo …>…</VectorInfo>
  …
  <Constraint …>…</Constraint>
  …
</PropertyInfo>
```

2.4.3.8 Constraint [Constraint]

Attributes:

- Name [n]: Identifies the constraint. Must be unique within the containing element.

Child elements: Constraint, Description (optional).
Description:
Constrains the behaviour of the corresponding Property. Usually the data type of the property is the same for all elements (i.e. the property is a scalar or simple vector) which means that the constraint constrains further the behaviour of this data type. In those special cases, where the property is made of several data types the constraint constrains the whole property, which must be taken into account when the semantic description of

the constraint is written. The constraint is a mechanism that can be used by the department designer and the semantics of each used constraint must be described in the component type description. PropertyInfo can have as many constraints as necessary. These that follow are the mutually exclusive constraints that are used by the system itself:

- StartTerminal: A Constraint you can add only to reference type PropertyInfo with allowed size two. The type of the terminal is defined as a sub-constraint and there can be only one optional type i.e. only one optional sub-constraint. The first element of the property vector contains an optional reference to the endTerminal property on the same level, i.e. connection. Empty string means no connection. The connection is further constrained by the reference rule mechanism (see ReferenceRule element). Note that the connection is not constrained by the reference rule mechanism if the constraint does not have a sub-constraint. The second element contains an optional reference to startTerminal on the lower level, i.e. mapping. Empty string means no mapping. If the property constrained by startTerminal is referred from the upper level (i.e. is mapped to the upper level), then it cannot have a connection element (i.e. the first element must be an empty string). It can still contain a mapping to the lower level. The mapping can only be made to a terminal of the same type.

- EndTerminal: A constraint you can add only to reference type PropertyInfo with allowed size two. The first value must always be empty. The type of the terminal is defined as a sub-constraint and there can be only one optional type, i.e. only one optional sub-constraint. The element contains an optional reference to endTerminal on the lower level, i.e. mapping. Empty string means no mapping. The mapping can only be made to a terminal of same type.

- PropertyMapping: A constraint you can add only to reference type PropertyInfo with allowed size one. The element contains an optional reference to the property on the lower lever. There cannot be an optional index part, i.e. the whole property is lifted to the upper level.

- Curve. Curve is a constraint you can add only to PropertyInfo the type of Vector and allowed size greater than one. There must always be at least two vector infos, even if they are the same (normally you can define only one vector info and let the default mechanism use this for all sub-vectors). The first sub-vector info describes the x-values for the curve. Its type must be short, integer, float or double, and its allowed size must be greater than one (at least two points to make a curve). The second sub-vector info describes the y-values for the curve. Its type must be short, integer, float or double, and its allowed size must be the same as with the first sub-vector. After the two compulsory sub-vector infos there can be an optional number of qualifier definitions. A qualifier definition is a scalar (allowed size is 1) and describes some property (for example measuring condition) for the curve. The meaning of the measuring conditions are usually explained in the Description element of the PropertyInfo element. The optional label attribute of the VectorInfo element can be used to make the description text a little more easily understandable. Also some clients can display the label and qualifier value with the curve.

- CurveSet. PropertyInfo marked with the curveSet constraint is a PropertyInfo definition that consists of one VectorInfo that consists of one VectorInfo definition which describes a curve structure (see above). All curves of the curveSet instance are curves of this type.

- ColumnTable: ColumnTable is a constraint you can add only to PropertyInfo whose VectorInfo type is vector and allowed size is three. The first sub-vector info describes row labels of the table. Its type must be string and its allowed size defines the maximum number of rows. The second sub-vector info describes the column labels of the table. Its type must be string and its allowed size defines the number of columns. The third vector info describes the column vectors. Its type must be vector and its size the number of columns. If all the column vectors are of the same type, then this third vector info needs only one sub-vector info because the other vector infos are generated by the default mechanism. The sub-vector infos must have the same allowed size maximum as the number of rows (each column vector has a value for each row). If the column vectors are made of different types then you must define the sub-vector info for each column vector so that you can define a different type for each vector.

- RowTable: RowTable is a constraint you can add only to PropertyInfo whose VectorInfo type is vector and allowed size is three. The first sub-vector info describes row labels of the table. Its type must be string and its allowed size defines the number of rows. The second sub-vector info describes the column labels of the table. Its type must be string and its allowed size defines the maximum number of columns. The third vector info describes the row vectors. Its type must be vector and size the number of rows. If all the row vectors are of the same type, then this third vector info needs only one sub-vector info because the other vector infos are generated by the default mechanism. The allowed size maximum of the sub-vector infos must be same as the number of columns (each row vector has a value for each column). If the row vectors are made of different types then you must define the sub-vector info for each row vector so that you can define a different type for each vector.

Example:

Example of PropertyMapping:

```
<PropertyInfo …>
  <VectorInfo type=”reference”/>
  <Constraint name=”propertyMapping”/>
</PropertyInfo>
```

Example of StartTerminal:

```
<PropertyInfo …>
  <VectorInfo type=”reference” allowedSize=”2”/>
  <Constraint name=”start Terminal”>
    <Constraint name=”flow”/>
  </Constraint>
</PropertyInfo>
```

In the above, the additional constraint tells us that only the connection rules that have flow as the start constraint apply to this connection. A typical rule for this connection would be a rule with flow as the start and end constraint.

Example of Curve:

```
<PropertyInfo …>
             <VectorInfo type=”vector” allowedSize=”3”>
    <VectorInfo type=”float” allowedSize=”666” label=”x-values”/>
    <VectorInfo type=”float” allowedSize=”666” label=”y-values”/>
    <VectorInfo type=”float” label=”condition 1”/>
  </VectorInfo>
  <Constraint name=”curve”/>
</PropertyInfo>
```

Example of CurveSet:

```
<PropertyInfo …>
  <VectorInfo type=”vector” allowedSize=”1..666”>
    <VectorInfo type=”vector” allowedSize=”3”>
      <VectorInfo type=”float” allowedSize=”666” label=”x-values”/>
      <VectorInfo type=”float” allowedSize=”666” label=”y-values”/>
      <VectorInfo type=”float” label=”condition 1”/>
    </VectorInfo>
  </VectorInfo>
  <Constraint name=”curve set”/>
</PropertyInfo>
```

Example of ColumnTable:

```
<PropertyInfo …>
  <VectorInfo type=”vector” allowedSize=”3”>
    <VectorInfo type=”string” allowedSize=”0..11” label=”row labels”>
```

```
      <Def i=”1”>row label 1</def>
      …
    </VectorInfo>
    <VectorInfo type=”string” allowedSize=”0..12” label=”column
labels”>
      <Def i=”1”>column label 1</def>
      …
    </VectorInfo>
    <VectorInfo type=”vector” allowedSize=”12” lable=”columns”>
      <VectorInfo type=”float” allowedSize=”11” label=”column”/>
    <VectorInfo/>
  </VectorInfo>
  <Constraint name=”rowTable”/>
</PropertyInfo>
```

```
<V>
  <V>row label 1~row label 2~…</V>
  <V>column label 1~column label 2~…</V>
  <V>
    <V>0 0…</V>
    …
  </V>
</V>
```

Example of RowTable:

```
<PropertyInfo …>
  <VectorInfo type=”vector” allowedSize=”3”>
    <VectorInfo type=”string” allowedSize=”0..11” label=”row labels”>
      <Def i=”1”>row label 1</def>
      …
    </VectorInfo>
    <VectorInfo type=”string” allowedSize=”0..12” label=”column
labels”>
      <Def i=”1”>column label 1</def>
      …
    </VectorInfo>
    <VectorInfo type=”vector” allowedSize=”11” label=”rows”>
      <VectorInfo type=”float” allowedSize=”12” label=”row”/>
    <VectorInfo/>
  </VectorInfo>
  <Constraint name=”rowTable”/>
</PropertyInfo>
```

Example of client specific constraints:

```
<PropertyInfo …>
  <VectorInfo type=”string”/>
  <Constraint name=”My Constraint”/>
  <Constraint name=”My Second Constraint”>
    <Constraint name=”General Properties”/>
  </Constraint>
  …
</PropertyInfo>
```

In the above, we have constrained the property info with “My Constraint”. In the component type description we would have to describe what this constraint means. The second constraint is further constrained by an additional constraint. In this case, the additional constraints seem to group the property into different groups. Probably other properties would have the same constraint with different additional constraints, such as Nuclear Properties and so on. The meaning of “My Second Constraint” and its additional constraints must also be described in the component type description. The server does not put any additional meaning to the usage of these constraints so it is up to the clients to honour the semantics described in the component type descriptions.

2.4.3.9 VectorInfo [VI]

Attributes:

- *Type [t]:* The type attribute tells the type of the vector elements. Each element must be of the same type. (However, note that if you want to implement for example a vector with both integers and floats as elements, you can do this. Define a VI the elements of which are VI elements of either the float or integer type with allowed size one.) The type attribute can be a primitive data type:

  - boolean: true or false, default = false.

  - short: 16 bit integer value, default = 0.

  - integer: 32 bit integer value, default = 0.

  - float: 4 byte IEEE (Institute of Electrical and Electronics Engineers) single precision floating point format, default = 0.

  - double: 8 byte IEEE double precision floating point format, default = 0.

  - string: a character string, same code set as with the XML document, default = “”.

  - file: a string value that defines the binary id of the file in question. To get/set the data of this file you can use the Get/SetBinaryData functions.

  - reference: a string value that defines the UUID of the referred object and may contain an optional index part (e.g. “UUID 1 2 3”). Empty string is a void pointer. The index starts from one (1) like Fortran arrays (and not from zero (0) like C arrays).

Or complex data type:

- vector: A vector. [☺]
- Enumeration type id: An instance of an enumeration type defined earlier.

- *Unit [u]:* Unit for the vector elements. Optional.
- *Min [min]:* Minimum value for the vector elements. Optional.
- *Max [max]:* Maximum value for the vector elements. Optional.
- *Label [l]:* Label for the vector. Optional. Can be used to mark different sub-vectors to make the description text more understandable. Also model explorer displays the label and qualifier value with the curve when the VectorInfo defines a curve (see Constraint).
- *AllowedSize [as] (optional):* Defines the limits for the size of the vector. Allowed size can be * (0 or more i.e. unlimited), n (exactly n), n..m (the range from n to m; n < m, m can be *). These primitive expressions can be separated with commas like this: 1,5,8..19, 35..45. This tells us that the size of the vector must be either one, five or from eight to 19, or 35 to 45 elements. (And of course if you have specified * any other expressions are redundant.) The default value is one (used when the attribute is missing). When the type is Vector and you have less VI elements than the allowed size defines, then all the undefined elements are the same type as the last defined element. (That is, the last defined VI element is used for all the undefined elements.)

Child elements: Default or VectorInfo (mutually exclusive)
Description:

VI defines how to create vector instances. Recursion is allowed (i.e. VI can contain VI elements). Default elements define the default values for all vector elements (can be used only if the vector elements are primitive data type elements). If min, max or unit attributes are not given, the default specified by the primitive data type is used. Note that when looking for the default value, automatic type conversion is used if the type is something else than a string. (Float and double are truncated when converting to integer types. Zero is false, not zero is true when converting integer types to boolean type. If the string does not start with digits, it is converted to zero when converting the string to integer types.) The instance of vector info is vector and it defines the values for the instance data. Depending on the type of the vector info the data is divided into groups:

Tilde separated. The data of a vector grouped to the value of the vector element and separated by a tilde mark (~ ASCII character 127). String and reference are tilde separated. If the vector has n elements it must have n – 1 separators.

Space separated. The data of a vector grouped to the value of the vector element and separated by a space. All other types except file, reference, string and vector are space separated. If a vector has n elements, it must have n – 1 separators.

Vector separated. Each Vector type Vector Info is represented by a Vector element in the instance side. Each enumeration type element is also vector separated.

Release Notes For Gallery Setup 1.0:

The reference type can not contain an optional index expression. The allowed size expression can not contain sub-expressions separated with commas.

Example:

Here is a simple definition for a vector of floats:

```
<VectorInfo type=”float” allowedSize=”10”>
    <Default index=”1”>1</Default>
    <Default index=”2”>2</Default>
    <…>
</VectorInfo>
=>
<Vector>1 2 …</Vector>
```

Here is another:

```
<VectorInfo type=”float” allowedSize=”10” unit=”m” min=”0” max=”100”>
    <Default index=”1”>1</Default>
    <Default index=”2”>2</Default>
    <Default index=”3” count=”7”>50</Default>
</VectorInfo>
=>
<Vector>1 2 50 50 …</Vector>
```

And finally a vector of vectors:

```
<Enumeration id=”ENUM-UUID” name=”MyEnum”>
    <AllowedValue>enum one</AllowedValue>
    <AllowedValue>euum two</AllowedValue>
    …
</Enumeration>
<VectorInfo type=”vector” allowedSize =”*”>
```

```
    <VectorInfo type=”float” allowedSize=”1”>
        <Default>1<Default>
    </VectorInfo>
    <VectorInfo type=”ENUM-UUID” allowedSize=”2”>
        <Default index=”1”>enum one<Default>
        <Default index=”2”>enum two<Default>
    </VectorInfo>
    <VectorInfo type=”vector” allowedSize=”*”>
        <VectorInfo>…</VectorInfo>
        …
    </VectorInfo>
    …
</VectorInfo>
=>
<V>
  <V>1</V>
  <V>
    <V>enum one</V>
    <V>enum two</V>
  <V>
  …
</Vector>
```

2.4.3.10 Default [Def]

Attributes:

- Index [i] (Optional): Index of the first element to which this default specification applies. (The index of the first element is one (1), of the second element two (2) and so on.) If missing , the default value is one.

- Count [c] (Optional): How many elements this default specification concerns. If missing, the default value is one.

Child elements: None.
Description:
Def defines the default value for the corresponding value elements in the property.

Example:
```
<Default index=”1” count=”1”>999.999</Default>
<Default index=”2” count=”2”>999.999</Default>
```

2.4.3.11 Component [C]

Attributes:

- Id [id]: UUID of the element.

- Name [n]: Name of the component. Must be unique within the containing element.

- TypeId [tid]: UUID of the component type of this component.

- Suc [s] (optional): If exists and true the component is structural. The component is structural if it has child components. This is mainly for the GetTree function.

- Tracked [t] (optional): If set and “true” then the set component and set property commands add a modified child element. The Set property command also checks that the property has property info with existing trackable attribute and it is true before adding the modified element. When the element is created first the modified element is added regardless of this attribute.

- Runnable [r](optional): If the component type is runnable then this attribute must exist. (Created by the server.) The value can be “simulating” or “frozen”. The Freeze and Simulate functions change this.

- Loadable [l] (optional): The component is loadable when this attribute exists. The value of the attribute can be “loaded_changed” “loaded_unchanged” or “unloaded”. Child elements of loadable components are accessible (in queries and so on) only when the component is loaded. The loadable component itself and its attributes are always accessible. The model server always creates the loadable attribute, the client can only react to it. Only loadable components support the Load, UnLoad and Save functions.

Child elements: Rights (optional), Property, Modified, and Description (optional).
Description:
C represents an instance of a component type. The modified element is added automatically by the server and it keeps track of modifications, i.e. change of attributes, adding and removing of child elements. The first modified element is the creator i.e. owner.

Release Notes For Gallery Setup 1.0:

Simulate, Freeze, Load, Unload, Save functionalities are not implemented.

Example:

```
<Component[C] id=…>
    <Modified[M] by=”…” at=”…”/>
    …
    <Property[P] name[n]=”Name” id=…>
        <Vector[V]>AA-195</Vector[V]>
    </Property[P]>
    …
</Component[C]>
```

2.4.3.12 Property [P]

Attributes:

- Id [id]: UUID of the element.

- InfoId [iid]: UUID of the corresponding property info element.

- Lifted [l]: Can be true or false. If true, then this property is lifted from the lower level and the info id points to the property info of that lower level components property info. If the lifted property is a value or mapping property then the value of the data vector points to the (original) property at the lower level. If the lifted property is a terminal then the second value of the data vector points to the (original) property at the lower level. If false, the property is described in the property info.

Child elements: Vector.
Description:
The logic with the Vector child element is that it defines the values of the corresponding vector info elements defined in the property info. In the property info the vector info and its allowed size, type, unit, min and max attributes are defined. Vector, an instance of this vector info is used in the property. Only the vector element needs to be specified, because all other values can be obtained from the definition. The mechanism to map the vector info and default elements to the vector elements is as follows: The first vector info element is mapped to the vector element. Other vector info elements are mapped to one or many vector elements depending on the type of the vector info (see vector info). Corresponding values are created for each default element. The allowed size limits the number of values you can use. Note that in the simplest case the property does not have to contain any values at all, in this case the value vector is formed entirely from its definition.

Release Notes For Gallery Setup 1.0:

Lifter attribute not implemented.

Example:
Here is an example of a typical property info and property:

```
<PropertyInfo name=”Pump speed” id=”…” necessity=”optional”>
  <VectorInfo type=”float/>
</PropertyInfo>
<Property infoId=”…” id=”…” lifted=”false”>
  <Vector>10</Vector>
</Property>
```

Here is the property info and its (instantiated) property, this defines a float vector of size 10. The last seven values are all 50:

```
<PropertyInfo …>
    <VectorInfo type=”Float” allowedSize=”10” min=”0” max=”100”
unit=”m”>
        <Default index=”1”>1</Default>
        <Default index=”2”>2</Default>
        <Default index=”3” count=”8”>3</Default>
    </VectorInfo>
</PropertyInfo>
<Property infoId=”UUID” id=”UUID” lifted=”false”>
    <Vector>1 2 3 3 3 3 3 3 3 3</Vector>
<Property>
```

Here is another example:

```
<PropertyInfo name=”huuhaa” id=”UUID” necessity=”implied”>
    <VectorInfo type=”Vector” allowedSize =”*”>
        <VectorInfo type=”float”>
            <Default>1</Default>
        </VectorInfo>
        <VectorInfo type=”float” allowedSize=”2”>
            <Default index=”1”>1</Default>
            <Default index=”2”>2</Default>
        </VectorInfo>
        <VectorInfo type=”Vector” allowedSize=”*”>
            <VectorInfo>…</VectorInfo>
            …
        </VectorInfo>
        …
    </VectorInfo>
</PropertyInfo>
<Property infoId=”UUID” id=”UUID” lifted=”false”>
  <Vector>
    <Vector>1</Vector>
    <Vector>1 2</Vector>
    <Vector>
        <Vector>…</Vector>
        …
    </Vector>
</Property>
```

2.4.3.13 TerminalRules [TRs]

Child elements: ReferenceRule
Description:
An element is a container for reference rules. Terminal rules are reference rules for references that are restricted only between the properties of components that are on the same hierarchical level in the same hierarchy (i.e. the components must have a common father component).

Release Notes For Gallery Setup 1.0:

Terminal rules not supported.

2.4.3.14 ReferenceRule [RR]

Attributes:

- StartSubConstraint [s]: Identifies the sub-constraint the startTerminal constraint must have for this rule to be applied. If the startTerminal constraint does not have a sub-constraint then no rules are applied. If the start (or end) sub-constraint is "*" then it will match any constraint (including no constraint).

Child elements: EndSubConstraint, **Description** (**optional**).
Description:
Describes all allowed end constraints for one start constraint.

Example:
```
<TerminalRules>
    <ReferenceRule startSubConstraint=”flow”>
        <EndSubConstraint name=”flow”/>
    </ReferenceRule>
</TerminalRules>
```
The above rule states that the property with a startTerminal constraint with a flow sub-constraint can be connected to the property with an endTerminal constraint with a flow sub-constraint. This would typically need the following property info descriptions:

```
<ComponentType …>
  …
  <PropertyInfo id=”start” …>
    <Constraint name=”startTerminal”>
        <Constraint name=”flow”/>
    </Constraint>
  </PropertyInfo>
  …
</ComponentType>

<ComponentType>
  …
  <PropertyInfo id=”end” …>
    <Constraint name=”endTerminal”>
        <Constraint name=”flow”/>
    </Constraint>
  </PropertyInfo>
  …
</ComponentType>
<Component …>
    …
    <Property id=”from” infoId=”start”>
        <Vector>to~</Vector>
    </Property>
    …
</Component>
<!—This component is on same hierarchy level→
<Component …>
    …
    <Property id=”to” infoId=”end”>
      <Vector>~</Vector>
    </Property>
    …
</Component>
```

Here is another example:

```
<TerminalRules>
    <ReferenceRule startSubConstraint=”flow”>
       <EndSubConstraint name=”flow” allowedIndex=”1..2,*”>
    </ReferenceRule>
</TerminalRules>
```

The above rule states that the property with a startTerminal constraint with a flow sub-constraint can be connected to the property with an endTerminal constraint with a flow sub-constraint. The allowed index attribute means that this is a reference to a value and restricts the value set. The format of the allowed index is the same as the allowed size attribute of VectorInfo. This would typically need the following property info descriptions:

```
<ComponentType …>
  …
  <PropertyInfo id=”start” …>
    <Constraint name=”startTerminal”>
        <Constraint name=”flow”/>
    </Constraint>
  </PropertyInfo>
  …
</ComponentType>
<ComponentType>
  …
  <PropertyInfo id=”end” …>
    <Constraint name=”endTerminal”>
        <Constraint name=”flow”/>
    </Constraint>
  </PropertyInfo>
  …
</ComponentType>
<Component …>
    …
    <Property id=”from” infoId=”start” lifted=”false”>
        <!—This is a reference to element 6. The mapping is void.→
        <Vector>to 2 3~</Vector>
    </Property>
    …
</Component>
<!—This component is on same hierarchy level→
<Component …>
  …
  <Property id=”to” infoId=”end”>
    <Vector>
      <Vector>1 2 3</Vector>
      <Vector>4 5 6</Vector>
    <Vector>
  </Property>
    …
</Component>
```

2.4.3.15 EndSubConstraint [ESC]

Attributes:

- Name [n]: Name of the allowed sub-constraint of the endTerminal constraint. Can be empty which means no constraint. Can be * which means anything goes.

- AllowedIndex [a] (optional): The allowed index attribute means that this a reference to a property value and restricts the set of target values. The format of the allowed index is the same as the allowed size attribute of VectorInfo.

- ComponentType [c] (optional): The required type of the owner component of the end property.

Description:

ESC describes the constraints for the end of the reference.

2.4.3.16 Enumeration [E]

Attributes:

- Id [id]: UUID of the element.

- Name [n]: Name of the enumeration. Must be unique within the containing type library. Cannot be any of the predefined type names (double, vector, etc.).

Child elements: AllowedValue [AV], Description (optional), Modified (optional), Rights (optional).
Description:

With E you can define enumeration types. These enumeration types can be used as VectorInfo types in the same way as primitive types (with the exception that enumeration types are type library specific). You can add Enumeration elements to type library categories and to their descending categories.

(See GetComponentType, GetEnumeration, GetTree, ModiEnumeration, RemoveEnumeration, SetEnumerationNew, SetEnumerationOld).

Example:

```
<Enumeration id="UUID" name="MyEnum">
    <AllowedValue>huu</AllowedValue>
    <AllowedValue>haa</AllowedValue>
    …
</Enumeration>
```

2.4.3.17 ClientExtension [CE]

Attributes:

- Id [id]: UUID of the element.
- Name [n]: Name of the extension.
- Type [t]: Type of the extension. Used for grouping the extensions to sets that have the same interface. The interface of the types is described in the Description element (if the provider of the client extension wants to let other clients than her own client to use the client extension in question). Note that each client extension that has the same type must have the same interface or confusion will follow.
- BinaryId [bid]: The binary id of the client extension.
- Att-n (optional): The client extension can have as many optional arguments as necessary. The meaning of the arguments is described in the Description element.
- ComponentTypeList [ctl] (optional): List of component type ids separated by a space.
- ServerExtensionList [sel] (optional): List of server extension ids separated by a space.

Child elements: Description (optional), Modified (optional), Rights (optional).
Description:
Model Explorer and other clients of model servers can extend the features that are available for them with a client extension. Client extension is a binary component that is run on the client machine. Its invocation mechanism is described in the Description element. The invocation of components with the same type is always the same. Client applications know which client extension types they support. The available extensions are retrieved with the GetTree function. Other functions manipulating the client extension are RemoveClientExtension and SetClientExtension. Client extension elements can only be in type libraries.

2.4.3.18 ServerExtension [SE]

Attributes:

- Id [id]: The UUID of the server extension.

- Name [n]: Name of the extension.

- FunctionName [fn]: Name of the function. Only digits (0-9) and letters (a-zA-Z) allowed, and name must start with letter. Each function requires a separate SE element. The function arguments are described by the Input- and OutputArgument elements.

- BinaryType [bt]: Describes the binary format of the component. Can be COM or DLL. COM binary type means that the component is a COM Automation component. DLL binary type means that the component is dynamic load library. The invocation mechanism is binary type specific.

- OperatingSystem [os]: Tells the operating system for which this component is meant. If same as the operating system where the Model Server is run then the component can be invoked. Allowed values are NT, W2K. Gallery Server's operating system is W2K (Windows 2000 Advanced server).

- Type [t]: Type of the extension. Can be searchEngine or other. Search engine is meant for extending the search capabilities of the server (i.e. offering special pump search algorithms and such). The type other is for all other uses.

- ProgramId [pid]: Identifier for the component. Might be needed when the extension method is invoked. For COM this is the programmatic id of the component. For DLL this is a descriptive name for the library.

- BinaryId [bid]: The binary id of the server extension. Several server extensions can have same binary id.

- TypeId [tid] (optional): UUID of component type. The server extension can process components of this type. Compulsory for search engines.

Child elements: Modified (optional), Rights (optional), InputArgument (optional), OutputArgument (optional).
Description:

With server extensions clients can extend the features of the Model Server. Server extension is a binary component that is run on the Model Server machine. The binary components can use (call) GQL functions through the GQL API (Gallery 2002) and thus manipulate the Model Server as necessary. The client can invoke the server extension with InvokeServerExtension function. The other functions manipulating server extensions are GetTree, GetServerExtension, RemoveServerExtension, SetServerExtension, and ListServerExtensionBinaries. Server extension elements can only be in type libraries.

Each server extension element describes one function of the corrseponding binary component. The first input argument of each function is always the GQL call context (Gallery 2002), search engines have additional compulsory type id argument. Rest of the input arguments are described by InputArgument elements. All output arguments are described by OutputArgument elements. By convention the first two InputArgument elements of the search engine must be “Department Id” and “Category Id”. These define a context for the search and filled in automatically by Model Explorer when user invokes search engines. The rest of input arguments are free. The first output argument must “Component Data”, a string containing data of component type followed by data of selected components in priority order according to the search criteria. If the components in question contain calculated data values then these must be included as optional properties to the component type in question. Rest of the output parameters must be arrays of any type. The length of the arrays must be the same as the number of returned components. The arrays contain the parameter values that the extension considers especially relevant for the search results.

Release Notes For Gallery Setup 1.0:

Only COM server extensions are supported.

Example:

Here is an IDL for simple binary component containing four functions, and corresponding server extension elements (five functions, the last function is described as search engine function and as a normal function):

```
import "oaidl.idl";
import "ocidl.idl";
[
    object,
    uuid(A7D6F093-4899-44AB-8DB3-6CB90F71EA19),
    dual,
    helpstring("ISimple Interface"),
    pointer_default(unique)
]
interface ISimple : IDispatch
{
    [id(1), helpstring("method method1")] HRESULT method1([in]long
aContext, [in]long ai, [out]long* api, [in]double ad, [out]double*
apd, [in]BSTR as, [out]BSTR* aps);

    [id(2), helpstring("method method2")] HRESULT method2([in]long
aContext, [out]BSTR* aps);
```

```
    [id(3), helpstring("method method3")] HRESULT method3([in]long
aContext, [in]SAFEARRAY(long)apIn, SAFEARRAY(long)* apOut,
[in]SAFEARRAY(double)apdIn, SAFEARRAY(double)* apdOut,
[in]SAFEARRAY(BSTR)apsIn, SAFEARRAY(BSTR)* apsOut);

    [id(4), helpstring("method method4")] HRESULT method4([in]long
aContext, [in]BSTR aTypeId, [in]BSTR aDepId, [in]BSTR aCatId,
[out]BSTR* apCompData);
};

[
    uuid(33623B8B-E8EE-476A-BE53-BE8AEFCBDB55),
    version(1.0),
    helpstring("ServerExtension 1.0 Type Library")
]
library SERVEREXTENSIONLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
         uuid(AA12BE15-DE22-4BB6-B91C-1437A1CA2B2F),
         helpstring("Simple Class")
    ]
    coclass Simple
    {
        [default] interface ISimple;
    };
};

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="AFBFDBAA9-BF46-41ca-B066-8BC77A644E77"
n="ServerExtension.Simple.method1" fn="method1" bt="COM" os="NT"
t="other" pid="ServerExtension.Simple">
   <IA n="Arg1" dt="integer" ne="required"/>
   <OA n="Arg2" dt="integer"/>
   <IA n="Arg3" dt="double" ne="required"/>
   <OA n="Arg4" dt="double"/>
   <IA n="Arg5" dt="string" ne="required"/>
   <OA n="Arg6" dt="string"/>
</SE>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="AFBFDBAAA-BF46-41ca-B066-8BC77A644E77"
n="ServerExtension.Simple.method2" fn="method2" bt="COM" os="NT"
t="other" pid="ServerExtension.Simple">
   <OA n="Output" dt="string"/>
</SE>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="A3B9A7675-4FB8-4d85-8C65-5B9537946E18"
n="ServerExtension.Simple.method3" fn="method3" bt="COM" os="NT"
t="other" pid="ServerExtension.Simple">
   <IA n="Arg1" dt="integer" ne="required" as="*"/>
   <OA n="Arg2" dt="integer" as="*"/>
   <IA n="Arg3" dt="double" ne="required" as="*"/>
   <OA n="Arg4" dt="double" as="*"/>
   <IA n="Arg5" dt="string" ne="required" as="*"/>
   <OA n="Arg6" dt="string" as="*"/>
</SE>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="A1A74114B-2CCF-4918-9B52-E81DF2C7E92A"
n="ServerExtension.Simple.method4" fn="method4" bt="COM" os="NT"
t="searchEngine" pid="ServerExtension.Simple" tid="AB1B8DB20-3BD9-
11d6-9B4B-00105A688D46">
   <IA n="Department Id" dt="string" ne="required"/>
   <IA n="Category Id" dt="string" ne="required"/>
   <OA n="Component Data" dt="string"/>
</SE>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="AF693144E-0157-448c-834A-D6A82029E833"
n="ServerExtension.Simple.method4a" fn="method4" bt="COM" os="NT"
t="other" pid="ServerExtension.Simple">
   <IA n="Type Id" dt="string" ne="required"/>
   <IA n="Department Id" dt="string" ne="required"/>
   <IA n="Category Id" dt="string" ne="required"/>
   <OA n="Component Data" dt="string"/>
</SE>
```

Here is another example:

```
    [id(1), helpstring("method search")] HRESULT search([in] long
aContext, [in] BSTR aTypeId,[in] BSTR aDepartment, [in] BSTR
aCategory, [in] double aDutypointflow, [in] double aDutypointhead,
[in] double aNpsha, [in] long aImpellerType, [in] long aFrequency,
[in] long abConstantspeed, [in] double aManualspeed, [in] double
aMaxspeed, [out] BSTR* apXMLComponents, [out] SAFEARRAY(BSTR)
*appName, [out] SAFEARRAY(double) *appImpDia, [out] SAFEARRAY(double)
*appSpeed, [out] SAFEARRAY(double) *appNpshr, [out] SAFEARRAY(double)
*appPowers, [out] SAFEARRAY(double) *appEfficiency);

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SE SYSTEM "serverExtension.dtd">
<SE id="a4dc4366f-31a3-4c4f-912a-1ee07df56ec3" n="PumpDesign.search"
fn="search" bt="COM" os="NT" t="searchEngine"
pid="PumpSelection.Pumps" tid="acad8264b-8df8-4488-b0b1-2c1db156f207">
   <IA n="Department Id" dt="string" ne="required"/>
   <IA n="Category Id" dt="string" ne="required"/>
   <IA n="Dutypoint flow" dt="double" ne="required" u="l/s" d="100"/>
   <IA n="Dutypoint head" dt="double" ne="required" u="m" d="20"/>
   <IA n="NPSHa" dt="double" ne="required" u="m" d="0"/>
   <IA n="Impeller type" dt="integer" ne="required" d="0"/>
   <IA n="Frequency" dt="integer" ne="required" u="Hz" d="50"/>
   <IA n="Constant speed?" dt="integer" ne="required" d="0"/>
   <IA n="Manual speed" dt="double" ne="required" u="rpm" d="0"/>
   <IA n="Max speed" dt="double" ne="required" u="rpm" d="0"/>
   <OA n="Component Data" dt="string"/>
   <OA n="Component name" dt="string" as="*"/>
   <OA n="Impeller diameter" dt="double" as="*" u="mm"/>
   <OA n="Speed" dt="double" as="*" u="rpm"/>
   <OA n="NPSHr" dt="double" as="*" u="m"/>
   <OA n="Power" dt="double" as="*" u="kW"/>
   <OA n="Efficiency" dt="double" as="*" u="%"/>
</SE>
```

2.4.3.19 InputArgument [IA]

Attributes:

- Name [n]: Name of the argument. Must be unique among the containing element.

- DataType [dt]: integer (4), double (8), string (?). The lengths of the data types are given in parenthesis. String is always passed as a reference and its length is given as a second argument.

- AllowedSize [as] (optional): If the argument exists then this argument is array. The format of the allowed size is the same as with vector info allowed size attribute. Array is always passed as reference and the actual length of the array is always passed as well.

- Necessity [ne]: Can be required or optional. (For COM components optional argument type is VT_EMPTY)

- Unit [u] (optional): Unit of the argument (e.g. kg/s).

- Default [d] (optional): Default value for the argument (e.g. 3.14).

Description:

Defines input arguments for the extension. The order of the input elements must be the same in the actual function calls and in Invoke function.

2.4.3.20 OutputArgument [OA]

Attributes:

- Name [n]: Name of the argument.

- DataType [dt]: Same as with input argument (see above). The difference here is that output arguments are always passed as references.

- Unit [u] (optional): Unit of the argument (e.g. kg/s).

Description:

Defines output arguments for the extension. The order of the output elements must be the same in the actual function calls and in Invoke function.

3. Gallery query language

3.1.1 Introduction to query functions

GQL functions are typed to be either obligatory or optional. Every model server must implement the obligatory functions. It is up to each server to decide which optional functions it will and will not implement, although the recommendation is that they are all implemented. Each client has to check if the optional function it uses is or is not implemented. The implementation of the optional functions must be static, i.e. if a function returns “Not implemented.” then it must always return the same message and vice versa.

The client sends its request and receives the response with the mechanism provided by the transport libraries mctl.dll and mstl.dll. The request is not a complete XML document because it cannot use the XML header (<?xml version=”1.0” encoding=”ISO-8859-1”?>). At the moment, the version must always be 1.0 and the encoding must always be ISO-8859-1 and thus it does not need to be transferred by each request. The request contains an element for each function:

```
<Request>
    <Function>
        <Parameter>Value</Parameter>
    </Function>
    …
</Request>
```

Each (first level) element is given a sequence number (starting from one) and the response to this request is a response element that tells if the function call succeeded or failed. All the functions are separate transactions, so between the function calls other clients can change the state of the server. If one function call does not succeed possible modifications made by thr previous functions are not removed. If one function call fails the following function calls are not executed.

The response status can be ok or failed. If the satus is ok then the response element will have return child elements for each function. If the status is failed then the response element will have return elements for each of those elements that completed successfully and an error element for the failed element. Each function call is identified by the sequence number:

```
<Response status=”ok”>
    …
    <Return seq=”666”>
        <Argument>Return data</Argument>
    </Return>
```

```
        …
</Response>
```
or

```
<Response status=”failed”>
    …
    <Return seq=”666”>
        <Argument>Return data</Argument>
    </Return>
    …
    <Error seq=”999”>Undefined name</Error>
</Response>
```

If the server encounters an error before it has started to process the functions the response can be something like this:

```
<Response status=”failed”>
    <Error>No Role.</Error>
</Response>
```

In the following function descriptions the function name is the header (e.g. AddUser) and the input aguments are described first (e.g. UserName). The input arguments are transferred to the child elements of the corresponding function element (e.g. <AddUser> <UserName>…). The order of the argument elements in the request must follow the order in the function description. The return arguments are described after the input arguments. The argument elements in the return element are in the same order as they are described in the following function descriptions. The order of the child elements of the return elements is free if not otherwise specified.

Note that in addition to the correct required role, you must have the proper access rights to the element before you can access it and to access information inside a department, you must belong to that department, i.e. your name must be among the user definitions of that department.

Even though the attribute names in the following descriptions start with a capital letter, in XML elements they always start with a lower case letter e.g. <Category typeLibrary=”…” …/>.

3.1.2 GetAccessRightsInElement

Gets the access rights in a given element.

Input arguments:
Department: UUID of the Department.

Id: UUID of the element.

Type: Type of the element. Can be Dep, Ca, E, or C.

Return arguments:
Returns access rights for the user who is the client.

Error values: Insufficient rights. Element does not exist.

Required role: ModelUser.
Type: Obligatory.

Example:

```
<GetAccessRightsInElement>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>ae394b899-681b-44b3-a976-db5f0f482e32</Id>
  <Type>Dep</Type>
</GetAccessRightsInElement>

<Return seq="1">
  <AccessRightsInElement access="write"/>
</Return>
```

3.1.3 GetAccessRightsInGroup

Gets the access rights in a given group.

Input arguments:
Department: UUID of the Department.

Group: Name of the group.

Return arguments:
Returns access rights for the user who is the client.

Error values: Insufficient role. Insufficient rights. Group does not exist.

Required role: ModelConfigurator.
Type: Obligatory.

Example:

```
<GetAccessRightsInGroup>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Group>Everyone</Group>
</GetAccessRightsInGroup>
```

```
<Return seq="1">
<AccessRightsInGroup access="none"/>
</Return>
```

3.1.4 GetAdministrationTree

Gets the users and groups of a department and their access rights.

Input arguments:

- Department: UUID of the Department.

Return arguments:
Returns user definition and group definition elements.

Error values: Illegal argument.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<GetAdministrationTree>
  <Department>adff97c6c-e8cf-41e9-bea3-ddd48600c612</Department>
</GetAdministrationTree>

<Return seq="1">
  <A>
    <UD n="VTT kernel developer"/>
    …
    <GD n="Everyone" cby="system">
      <AR n="VTT kernel developer" a="read"/>
      …
      <U n="VTT kernel developer"/>
      …
    </GD>
  </A>
</Return>
```

3.1.5 GetAll

Gets all client extensions and server extensions. Returns only element attributes.

Input arguments:

- Department: UUID of the Department.

- ElementType: Defines the type of element that can be fetched. Can be CE, SE.

Return arguments:
Returns all the given elements.

Error values: Illegal argument.

Required role: ModelUser.
Type: Obligatory.

Example:
```
<GetAll>
  <Department>adff97c6c-e8cf-41e9-bea3-ddd48600c612</Department>
  <ElementType>CE</ElementType>
</GetAll>

<Return seq="1">
  <CE                        id="a38aab40d-b7ec-4da6-89df-e4804348d620"
n="CentrifugalPumpDialog"        t="componentDialog"        bid="5.ocx"
pid="CentrPumpAxProj1.CentrPumpAx" ctl="acad8264b-8df8
-4488-b0b1-2c1db156f207" sel="" >
  </CE>
  …
</Return>
```

3.1.6 GetAllowedTerminals

Gets all allowed end (or start) properties for the given start (or end) property.

Input arguments:
- Department: UUID of the Department.

- Property: An UUID of the property. The property must have either a start- or an endTerminal constraint.

- Level: 0 or 1. If zero then returns the allowed terminals that can be connected to the given terminal property according to the reference rules. The allowed terminals must be on the same level as the given terminal property and must not be referred to by terminal mappings. (If a terminal is mapped it can only be connected on the upper level.) In addition, the returned startTerminals cannot be connected (i.e. the first element of the property must be a null reference) and the endTerminals cannot be connected (i.e. no startTerminal on the same level can refer to that endTerminal). In the case of end terminals instead of an <ART> element an <ARF> element is returned. If one, then the system returns the allowed terminals (in the <AMP> element) that can be mapped, i.e. that belong to the child components of the component owning the given property, that are not connected, and have the same type as the given property.

- ExtendedPropertyInfo: See GetComponent for description.

- ExtendedVerticalInfo: See GetComponent for description.

- ExtendedHorizontalInfo: See GetComponent for description.

Return arguments:
Returns all possible end points for a given property.

Error values: Illegal argument.

Required role: ModelConfigurator**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Returns all start or end terminals without checking if they are connected or not. Terminal rules are not checked. Extended arguments are not implemented.

Example:

```
<GetAllowedTerminals>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Property>a25c47964-2cf8-4331-a7bb-8968bf569488</Property>
  <Level>0</Level>
  <ExtendedPropertyInfo>false</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>false</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>false</ExtendedHorizontalInfo>
</GetAllowedTerminals>

<Return seq="1">
  <C id="a962861d9-d8b6-41ad-97a6-a21df745fcb1" n="pipe1"
tid="a3dde37ea-0b23-4f1e-a6ca-0d60d8e2221b" t="false" r="not runnable"
l="not loadable" s="false" >
    <P id="a25c47964-2cf8-4331-a7bb-8968bf569488" iid="ae9d815e0-d04f-
49b9-81de-061999963775" >
    <ART>
      <P id="a5592dfe9-0818-4e39-8b46-12d0acd60b46" iid="a24759a10-
dc4e-4854-9ecd-57e79f870ba3" >
        <PI id="a24759a10-dc4e-4854-9ecd-57e79f870ba3" n="MODULE_NAME"
ne="implied" >
          <VI t="reference" as="2"/>
          <Constraint n="endTerminal">
            <Constraint n="endReference"/>
          </Constraint>
        </PI>
        <C id="a3857f39e-3451-4040-8378-6ebdcaeda5c2" n="point1"
tid="a2db9f2a0-9cb2-4546-b281-82bcb2861512" t="false" r="not runnable"
l="not loadable" s="false" >
        </C>
        <V>~</V>
      </P>
      …
    </ART>
    <V>~</V>
    </P>
  </C>
</Return>
```

3.1.7 GetBinaryData

Gets the binary data. If the request contains the GetBinaryData command it can not contain any other commands!

Input arguments:

- Department: UUID of the Department.
- BinaryId: The id of the binary data.

Return arguments:
Returns the data of the given binary id. The binary data is transferred after the response element.

Error values: Illegal argument. Insufficient role.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetBinaryData>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>6.ocx</Id>
</GetBinaryData>

<Return seq="1">
</Return>
\0…(The binary data.)
```

3.1.8 GetCategoryId

Gets the category id from a path expression. (Implementation note: This maximally slow operation was added to the set of operations afterwards because of heavy pressure from the model server programmers.)

Input arguments:

- Path: Category path expression. Made of element names separated by a slash e.g. Department name/ /Category name/Subcategory name.

Return arguments:

Returns the category id of the given category path.

Error values: Illegal argument. Insufficient role.

Required role: KernelDeveloper**.**

Type: Obligatory.

Example:

```
<GetCategoryId>
  <Path>…</Path>
</GetCategoryId>

<Return seq="1">
  <Ca id="…"/>
</Return>
```

3.1.9 GetComponent

Gets the component data by component id. In addition to pure component data you can also choose to get extended data for the component. Components you do not have access rights for are silently discarded.

Input arguments:

- Department: UUID of the Department.

- Id: The component identifier.

- Level: If zero then the component and its properties are returned. If one then the sub-components and their properties are returned.

- ExtendedPropertyInfo: If true then all the component properties will contain an additional property info element. This does not affect the property info elements of other extended data elements. The additional property info is the property info referred to by the info id attribute of the property. For all properties the property info is the property info referred to by the info id attribute of the property. If the vector type is enumerator the property info will contain the allowed values of that enumeration.

- ExtendedVerticalInfo: If true then the properties will contain additional navigation information. This does not affect the property elements of other extended data elements. This navigation information is made of the next property (NP), last property (LP), previous property (PP) and first property (FP). Each element contains a property element containing additional property info and a component element. The component element contains only the id and name attributes. The next property is the next lower property (property in the next lower level) in a vertical property chain. The last property is the lowest property in the chain. If the next and last are the same, only the next one is given. It the property is the last then both NP and LP are missing. The previous property is the next upper property. The first property is the topmost property. If the previous and first are the same then only the previous one is given. If the property is the topmost then both the previous and the first property are missing.

- ExtendedHorizontalInfo: If true then the end terminal properties and lifted end terminal properties will contain an additional reference from the (RF) element that contains a property with the same information as the extended vertical info elements. Note that the RF element contains only those start terminal properties that are on the same level. If you want all start terminals that refer to the end terminal you must navigate the mapping/lifting chain and request it from each level. The start terminal properties and lifted start terminal properties will contain an additional reference to the (RT) element which contains the same information as the RF element. Note that because the start terminal can only be connected on the top level only the top-level property of the mapping/lifting chain will contain the RT element.

Return arguments:
Returns component data in XML format. Returns component attributes and properties.

Error values: Component does not exist. Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelUser**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

The lifting mechanism and the mentioned lifted terminal properties are not implemented.

Example:
```
<GetComponent>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>af9fae87e-a0c0-44b8-9161cc8395fd13d3</Id>
  <Level>0</Level>
  <ExtendedPropertyInfo>false</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>true</ExtendedHorizontalInfo>
</GetComponent>

<Return seq="2">
  <C id="a962861d9-d8b6-41ad-97a6-a21df745fcb1" n="pipe1"
tid="a3dde37ea-0b23-4f1e-a6ca-0d60d8e2221b" t="false" r="not runnable"
l="not loadable" s="false" >
    <P id="a59fbc2f7-87f2-4569-907a-302de61da8f7" iid="aadef56c6-ed49-
4c22-ad79-1a412b4ef09b" >
      <V>1</V>
    </P>
    …
  </C>
</Return>

<GetComponent>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Level>0</Level>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>true</ExtendedHorizontalInfo>
</GetComponent>

<Return seq="1">
    <Component id="…" name[n]="…" typeId[tid]="…" …>
      <!—No description, modified or rights elements→
      <!—No sub components.→

      <!—Normal value property.→
      <Property id=”…” infoId[iid]=”…” lifted=”false”>
        <Vector>10</Vector>
        <PI>…</PI>
      </Property>

      <!--Mapping property.→
      <P …>
        <V>…</V>
        <PI …>
          <VI>…</VI>
          <Constraint>…</Constraint>
        </PI>
        <NP>
          <P …>
```

```
            <V>…</V>
            <PI …>
              <VI>…</VI>
              <Constraint>…</Constraint>
            </PI>
            <C id="…" n="…"/>
          </P>
        </NP>
        <!—Optional LP, PP, FP properties→
      </P>

           <!—Mapped/lifted start terminal property.→
      <!—Otherwise same as mapping, put also RT element.→

           <!—Mapped/lifted end terminal property.→
      <!—Otherwise same as mapping, put also RF element.→
    </Component>
  </Return>
</Response>
```

3.1.10 GetComponentId

Gets the component id from a path expression. (Implementation note: This maximally slow operation was added to the set of operations afterwards because of heavy pressure from the model server programmers.)

Input arguments:

- Path: Component path expression. Made of element names separated by a slash e.g. Department name/ /Category name/Subcategory name/Component name/Sub component name.

Return arguments:
Returns the component id of the given component path.

Error values: Illegal argument. Insufficient role.

Required role: KernelDeveloper**.**
Type: Obligatory.

Example:

```
<GetComponentId>
  <Path>…</Path>
</GetComponentId>

<Return seq="1">
  <C id="…"/>
</Return>
```

3.1.11 GetComponentType

Fetches the component type data by id.

Input arguments:

- Department: UUID of the Department.
- GetEnums: If true then the enumerators used by this component type are also fetched.
- Id: Id of the component type.

Return arguments:
Returns the component type data (see example).

Error values: Insufficient rights.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetComponentType>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <GetEnums>false</GetEnums>
  <Id>a15afdc00-83b9-4532-8fc3-717759864318</Id>
</GetComponentType>

<Return seq="1">
  <CT  id="a15afdc00-83b9-4532-8fc3-717759864318"  n="AB_CONVERTER_TYPE"
l="false" r="false" lo="false" >
    <PI   id="af474d9cc-e8c0-436e-96a8-d829f8d68efc"   n="MODULE_NAME"
ne="implied" >
      <VI t="reference" as="2"/>
      <Constraint n="endTerminal">
        <Constraint n="endReference"/>
      </Constraint>
    </PI>
    …
</Return>
```

3.1.12 GetComponentTypeData

Gets the component type and its components under the given category. This command is mainly meant for search engine server extensions.

Input arguments:

- Department: UUID of the Department.
- Component type: UUID of the component type.
- Category: UUID of the category. If null then gets all components of the given type.

Return arguments:

Returns the component type and components of that type under the given category.

Error values: Illegal argument. Insufficient rights.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetComponentTypeData>
  <Department>UUID</Department>
  <ComponentType>UUID</ComponentType>
  <Category>UUID</Category>
</GetComponentTypeData>

<Return seq='1'>
  <ComponentType[CT] name[n]="…" …>
   …
  </ComponentType[CT]>
  <Component[C] name[n]="…" …>
   …
  </Component[C]>
</Return>
```

3.1.13 GetDescription

Gets the description element.

Input arguments:

- Department: UUID of the Department.
- Id: UUID of the description element.

- Type: Type of the element. Can be Dep, Ca, E, CE, SE, C, CT.

Return arguments:
Returns the description element if it exists.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetDescription>
  <Department>adff97c6c-e8cf-41e9-bea3-ddd48600c612</Department>
  <Id>a207ef66c-b0a3-46db-9729dd898a99298a</Id>
  <Type>Ca</Type>
</GetDescription>

<Return seq="1">
<D><![CDATA[This is the type library of Gallery department.]]></D>
</Return>
```

3.1.14 GetDocument

Gets the documentation. If the request contains the GetDocument command it can not contain any other commands!

Input arguments:
- Department: UUID of the Department.

- Id: Id of the element to document.

- Type: Type of the element. Can be Dep, Ca, E, CE, SE, C, CT.

- Recursive: If true then the document will contain a description of the child elements.

Return arguments:
Returns the document in html format (see example).

Error values: Insufficient rights. Illegal argument.

Required role: ModelUser**.**
Type: Obligatory.

```
<GetDocument>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>ae394b899-681b-44b3-a976-db5f0f482e32</Id>
  <Type>Dep</Type>
  <Recursive>true</Recursive>
</GetDocument>

<Return seq="1">
</Return>
\0…(The html document.)
```

3.1.15 GetEnumeration

Fetches the enumeration data by id.

Input arguments:

- Department: UUID of the Department.

- Id: Id of the enumeration.

Return arguments:
Enumeration data (see example). Does not return the enumeration id!

Error values: Insufficient rights. Illegal argument. Department does not exist. Enumeration does not exist.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetEnumeration>
  <Department>adff97c6c-e8cf-41e9-bea3-ddd48600c612</Department>
  <Id>a68466b78-5516-4acc-8476-af69b8c267be</Id>
</GetEnumeration>

<Return seq="1">
<E  id="a68466b78-5516-4acc-8476-af69b8c267be"  n="CrossBaffleFlowModel"
>
  <AV>s-s (side to side)</AV>
  <AV>o-u (upp to down)</AV>
  <AV>s-s</AV>
  <AV>Other</AV>
</E>
</Return>
```

3.1.16 GetModel

Gets the component data by component id. The components you do not have rights to will be silently discarded.

Input arguments:

- Department: UUID of the Department.

- Id: The component identifier.

Return arguments:
Returns the component attributes, properties, sub-components, description, modified and rights elements. The properties of the component are returned before the child components so the lifting/mapping chains of the properties are encountered in a top to down order.

Error values: Component does not exist. Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelUser**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

The lifting mechanism and the mentioned lifted terminal properties are not implemented.

Example:
```
<GetModel>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>af9fae87e-a0c0-44b8-9161-cc8395fd13d3</Id>
</GetModel>

<Return seq="1">
  <C id="af9fae87e-a0c0-44b8-9161-cc8395fd13d3" n="MyComponent"
tid="aef39de5c-2838-4287-9d1a-f5d8b0effc5e" t="false" r="not runnable"
l="not loadable" s="true" >
    <C id="a3857f39e-3451-4040-8378-6ebdcaeda5c2" n="point1"
tid="a2db9f2a0-9cb2-4546-b281-82bcb2861512" t="false" r="not runnable"
l="not loadable" s="false" >
      <P id="a7fb4a8a7-01d1-4857-87a3-cbf0bfa5382c" iid="a15fdd4a6-
df06-4f2a-b054-05fb660d5414" >
        <V>1</V>
      </P>
      …
    </C>
    …
  </C>
</Return>
```

3.1.17 GetModifiedTree

Gets the modified elements from the given element and optionally from its child elements.

Input arguments:

- Department: The UUID of the department.
- Parent: The element UUID (can be category, component type, or component). If null then the department is used as the element.
- Type: Type of the parent. Can be Dep, Ca, E, CE, SE, C, or CT.
- Levels: Number of child levels to return (i.e. if 0 modified elements are returned).

Return arguments:
Returns the modified elements.

Error values: Illegal argument.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<GetModifiedTree>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Parent>af96630ec-1dc2-4630-9747-31f9d5358873</Parent>
  <Type>Ca</Type>
  <Levels>0</Levels></GetModifiedTree>

<Return seq="1">
  <Ca   id="af96630ec-1dc2-4630-9747-31f9d5358873"   n="Model   Library"
s="true" >
    <M by="VTT kernel developer" at="2002-05-13 14:25:39.888">
      <![CDATA[Hilipati hippan. Nain se homma etenee.]]>
    </M>
  </Ca>
</Return>
```

3.1.18 GetProperties

Gets the properties. You must have read rights to the owner component.

Input arguments:

- Department: UUID of the Department.

- Parent: An UUID of the parent component. The parent component is the component that owns the propertyMapping properties and the sub-components that contain the set of possible properties.

- Level: 0 or 1. If zero then returns the propertyMapping properties and the lifted value and mapping properties of the parent component. If one then returns the properties of the child components which do not have the start/endTerminal constraint.

- ExtendedPropertyInfo: See GetComponent for description.

- ExtendedVerticalInfo: See GetComponent for description.

Return arguments:
The property element(s) in XML format.

Error values: Illegal argument. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Lifting mechanism and the mentioned lifted values not implemented.

Example:

```
<GetProperties>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Parent>a962861d9-d8b6-41ad-97a6-a21df745fcb1</Parent>
  <Level>1</Level>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
</GetProperties>

<Return seq="1">
  <C id="a962861d9-d8b6-41ad-97a6-a21df745fcb1" n="pipe1"
tid="a3dde37ea-0b23-4f1e-a6ca-0d60d8e2221b" t="false" r="not runnable"
l="not loadable" s="false" >
```

```
    <P id="a59fbc2f7-87f2-4569-907a-302de61da8f7" iid="aadef56c6-ed49-
4c22-ad79-1a412b4ef09b" >
      <PI id="aadef56c6-ed49-4c22-ad79-1a412b4ef09b" n="FLOW MODEL"
ne="implied" >
        <VI t="a04C57C7B-4AB9-4840-B313-36B685AD564F" as="1"
l="PI12_ACCURACY_LEVEL">
          <Def c="1">1</Def>
        </VI>
      </PI>
      <V>1</V>
    </P>
    …
  </C>
</Return>

<GetProperties>
  <Department>UUID</Department>
  <Parent>…</Parent>
  <Level>1</Level>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
</GetProperties>

<Return seq="1">
  <C …>
      <!—Nothing else but terminal properties.→
      <P id=”…” name=”…” lifted=”false”>
        <PI>…</PI>
        <V>…</V>
             <NP>
          <P…>
             <PI>…</PI>
             <C id=”…” n=”…”/>
             <V>…</V>
          </P>
        <NP>
        <!—These are like NP→
        <LP>…</LP>
        <PP>…</PP>
        <FP>…</FP>
      </P>
    </C>…
</Return>
```

3.1.19 GetPropertiesByInfo

Gets the properties by component id and property info id. Note that if there are lifted properties then it can be possible that there are many properties for one property info..

Input arguments:

- Department: UUID of the Department.

- Component: UUID of the component.

- PropertyInfo: UUID of the property info.

Return arguments:
The property element(s) in XML format.

Error values: Illegal argument. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Lifting mechanism and the mentioned lifted properties not implemented.

Example:

```
<GetPropertiesByInfo>
  <Department>UUID</Department>
  <Component>UUID</Component>
  <PropertyInfo>UUID</PropertyInfo>
</GetPropertiesByInfo>

<Return seq="1">
  <P id=”…”>
  …
</Return>
```

3.1.20 GetProperty

Gets the property.

Input arguments:
- Department: UUID of the Department.
- Id: UUID of the property.
- ExtendedPropertyInfo: See GetComponent for description.
- ExtendedVerticalInfo: See GetComponent for description.
- ExtendedHorizontalInfo: See GetComponent for description.

Return arguments:
The asked property. The Vector [V] element inside the Property [P] element is the last element so that the necessary data for handling the V element is available when parsing the result with a sax parser. Also the property info elements are immediately before the

V element so that the last PI element encountered with the sax parser before the V element is guaranteed to be the property info of the vector element.

Error values: Illegal argument. Insufficient role.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<GetProperty>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>a906cf564-3322-445c-a3ef-983d3be5c5b1</Id>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>false</ExtendedHorizontalInfo>
</GetProperty>

<Return seq="1">
  <C        id="a962861d9-d8b6-41ad-97a6-a21df745fcb1"        n="pipe1"
tid="a3dde37ea-0b23-4f1e-a6ca-0d60d8e2221b" t="false" r="not runnable"
l="not loadable" s="false" >
    <P id="a906cf564-3322-445c-a3ef-983d3be5c5b1" iid="a28937bbf-458a-
4d84-a0d3-f05f7fb622d0" >
      <PI   id="a28937bbf-458a-4d84-a0d3-f05f7fb622d0"   n="NUMBER   OF
CALCULATION NODES IN SIDE THE PIPE" ne="implied" >
        <VI t="integer" as="1" min="0" l="PI12_VOLUME_NUMBER"/>
      </PI>
      <V>0</V>
    </P>
  </C>
</Return>

<GetProperty>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedHorizontalInfo>true</ExtendedHorizontalInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
</GetProperty>

<Return seq="1">
<C id=”…” n=”…”>
  <P id=”…” iid=”…” lifted=”false”>
    <NP>
      <P…>
```

```
        <C id=”…” n=”…”/>
        <PI>…</PI>
        <V>…</V>
      </P>
    <NP>
    <!—These are like NP→
    <LP>…</LP>
    <PP>…</PP>
    <FP>…</FP>
    <RF>…</RF>
    <RT>…</RT>
    <PI>…</PI>
    <V>…</V>
  </P>
</C>
</Return>
```

3.1.21 GetPropertyId

Gets the property id from the property path expression. (Implementation note: This is a maximally slow operation added to the set of operations afterwards because of heavy pressure from the model server programmers.)

Input arguments:

- Path: Property path expression. Made of element names separated by a slash e.g. Department name/ /Category name/Subcategory name/Component name/Property name.

Return arguments:
Property id.

Error values: Illegal argument. Insufficient role.

Required role:
KernelDeveloper.
Type: Obligatory.

```
<GetPropertyId>
  <Path>…</Path>
</GetPropertyId>

<Return seq="1">
  <P id=”…”/>
</Return>
```

3.1.22 GetRole

Gets the role of the user calling this function.

Input arguments:
None.

Return arguments:
Returns the role of the user. Can be KernelDeveloper, Provider, ModelConfigurator (ModelDeveloper in implementation), or ModelUser.

Error values: Internal error.

Required role: ModelUser**.**
Type: Obligatory.

Example:
```
<GetRole/>

<Return seq="1">
  <Role n="KernelDeveloper"/>
</Return>
```

3.1.23 GetServerExtension

Gets the server extension element and its sub-elements.

Input arguments:
- Department: UUID of the Department.

- Id: UUID of the server extension.

Return arguments:
Returns the server extension and its sub-elements.

Error values:
Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetServerExtension>
  <Department>adff97c6c-e8cf-41e9-bea3-ddd48600c612</Department>
  <Id>a9c05282c-2fcc-4fea-b0a2-bb9edd8b4de6</Id>
</GetServerExtension>
```

```
<Return seq="1">
<SE   id="a9c05282c-2fcc-4fea-b0a2-bb9edd8b4de6"   n="HEXDesign.search"
fn="search"   bt="COM"   os="NT"   t="searchEngine"   pid="HEXDesign.HEX"
bid="9.dll" tid="aab7276fd-0e3f-4d6c-bc9e-52a17c5d1a41" >
<IA n="Department Id" dt="string" ne="required"/>
…
<OA n="Component Data" dt="string"/>
…
</SE>
</Return>
```

3.1.24 GetSkeletonDepartment

Gets the data necessary to copy a department and its categories to another model server.

Input arguments:

- Department: UUID of the Department.

Return arguments:

Returns the department and its categories.

Error values: Illegal argument.

Required role: KernelDeveloper**.**

Type: Obligatory.

Example:

```
<GetSkeletonDepartment>
  <Department>UUID</Department>
</GetSkeletonDepartment>

<Return seq="1">
<Department[Dep]…>
  <Category[Ca] …>…</Category[Ca]>
</Department>
</Return>
```

3.1.25 GetTerminalRules

Gets the terminal rules of a given type library. (Implementation note: At the moment Terminal Rules are not used for anything!)

Input arguments:

- Department: UUID of the Department.
- Id: UUID of the type library.

Return arguments:
Returns the terminal rules of a given type library.

Error values: Illegal argument. Insufficient role.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:
```
<GetTerminalRules>
  <Department>ae394b899-681b-44b3-a976-db5f0f482e32</Department>
  <Id>ac81e1fad-769e-407d-96c9-637b4fbb881c</Id>
</GetTerminalRules>

<Return seq="1">
<TRs/>
</Return>
```

3.1.26 GetTerminals

Gets all terminal properties (properties that have either the startTerminal or endTerminal constraint). Properties to whose owner component you do not have read rights are silently filtered out.

Input arguments:

- Department: UUID of the Department.
- Parent: If the level is zero, the UUID of the component. If the level is one the UUID of the parent component. The parent component is the component that owns the components that contain the set of possible properties.
- Level: 0 or 1. If zero then returns the terminals of the given component. If one then returns terminals that belong to child components of the given component.
- ExtendedPropertyInfo: See GetComponent for description.
- ExtendedVerticalInfo: See GetComponent for description.

- ExtendedHorizontalInfo: See GetComponent for description.

Return arguments:
All possible end points for connections.

Error values: Illegal argument. Insufficient role.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<GetTerminals>
  <Department>ae394b899-681b-44b3-a976db5f0f482e32</Department>
  <Parent>af9fae87e-a0c0-44b8-9161-cc8395fd13d3</Parent>
  <Level>1</Level>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>false</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>false</ExtendedHorizontalInfo>
</GetTerminals>

<Return seq="1">
<C        id="a3857f39e-3451-4040-8378-6ebdcaeda5c2"        n="point1"
tid="a2db9f2a0-9cb2-4546-b281-82bcb2861512" t="false" r="not runnable"
l="not loadable" s="false" >
  <P  id="a5592dfe9-0818-4e39-8b46-12d0acd60b46"  iid="a24759a10-dc4e-
4854-9ecd-57e79f870ba3" >
    <PI   id="a24759a10-dc4e-4854-9ecd-57e79f870ba3"   n="MODULE_NAME"
ne="implied" >
      <VI t="reference" as="2"/>
      <Constraint n="endTerminal">
        <Constraint n="endReference"/>
      </Constraint>
    </PI>
    <V>~</V>
  </P>
</C>
…
</Return>


<GetTerminals>
  <Department>UUID</Department>
  <Parent>…</Parent>
  <Level>0</Level>
  <ExtendedPropertyInfo>true</ExtendedPropertyInfo>
  <ExtendedVerticalInfo>true</ExtendedVerticalInfo>
  <ExtendedHorizontalInfo>true</ExtendedHorizontalInfo>
</GetTerminals>

<Return seq="1">
<C…>
  <!—Nothing else but terminal properties.→
  <P id=”…” name=”…” lifted=”false”>
    <NP>
      <P…>
        <C id=”…” n=”…”/>
        <PI>…</PI>
        <V>…</V>
      </P>
    </NP>
    <LP>…</LP>
    <PP>…</PP>
    <FP>…</FP>
    <RT>…</RT>
    <RF>…</RF>
    <PI>…</PI>
    <V>…</V>
  </P>
</C>
…
</Return>
```

3.1.27 GetTree

Gets the departments, categories, component types, components, enumeration and extensions. Does not return the terminal rules (see Get/SetTerminalRules).

Input arguments:

- Expand: UUID of the element you want to expand and the type of the element. The type can be Dep, Ca, E, or C. Those elements you give as argument are expanded (i.e. their sub-elements are also returned). Otherwise they are not expanded. The id hierarchy you give as an argument defines the return hierarchy. If the expand child id already belongs to the child list of the parent it is returned only once. In this case also the child element is expanded as asked. If the expand child id does not belong to the child list of the parent then the parent will have an extra child element containing the expanded element. If the given expand element id does not exist the server will silently ignore it. If the root expand element id is empty (“”) and the type is Root then all the departments are given as child elements.

Return arguments:

Returns the department, rights, category, component type and component elements, and the first Modified element (i.e. the owner). The enumeration elements are also returned but not their data.

Error values: Illegal argument.

Required role: ModelUser**.**
Type: Obligatory.

Example:

```
<GetTree>
<Expand id="" t="Root">
           <Expand id=”adff97c6c-e8cf-41e9-bea3-ddd48600c612” t=”Dep”/>
</Expand>
</GetTree>

<Return seq="1">
<Dep id="adff97c6c-e8cf-41e9-bea3-ddd48600c612" n="Gallery Department"
s="true">
  <M by="VTT kernel developer" at="2002-05-13 14:18:02.567">
    <![CDATA[Once upon a time.]]>
  </M>
  <R n="Everyone" a="read"/>
  <Ca   id="a207ef66c-b0a3-46db-9729-dd898a99298a"   n="Gallery   Type
Library" tl="Type Library" s="true" >
    <M by="VTT kernel developer" at="2002-05-13 14:18:02.577">
      <![CDATA[Once upon a time.]]>
    </M>
    <R n="Everyone" a="read"/>
  </Ca>
  …
</Dep>
…
</Return>
```

3.1.28 GetTypeLibrary

Gets the data necessary to copy a department and its type library to another model server.

Input arguments:
- Department: UUID of the Department.

Return arguments:
Returns the department, type library and its categories.

Error values: Illegal argument.

Required role: KernelDeveloper**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Not implemented.

Example:

```
<GetTypeLibrary>
  <Department>UUID</Department>
</GetTypeLibrary >

<Return seq="1">

  <Category[Ca] tl=”true” …>

    <Category[Ca]…>…</Category>

  </Category>

</Return>

\0…(The binary data.)
```

3.1.29 InvokeServerExtension

Calls the installed server extension.

Input arguments:

- Department: UUID of the Department.
- Id: UUID of the Server Extension.
- Input arguments: Name and value for each given input argument. If the allowed Size attribute is defined then the values are packed, separated the same way as the vector element data. (See Vector.) The server matches the input arguments by name, so optional input arguments are needed only when necessary (i.e. you do not have to give empty values to optional input arguments).

Return arguments:

- Output arguments: Name and value for each output argument. The order of the output arguments is the same as in the order of the output argument elements in the server extension element.

Error values: Insufficient role. Insufficient rights.

Required role: ModelUser.
Type: Obligatory.

Example:

```
<InvokeServerExtension>
  <Department>UUID</Department>
  <Id>…</Id>
  <IA n=”arg1” ne=”required” as=”3”>1 3 5</InputArgument>
</InvokeServerExtension>
</Request>

<Return seq=”1”>
  <OA>1 3 5</OA>
</Return>
```

3.1.30 ListClientExtensions

Lists the client extensions suitable for a given component type or server extension.

Input arguments:

- Department: UUID of the Department.

- Id: UUID of the component type or server extension depending on the type argument.

- Type: Can be “CT” or “SE”. If CT then lists all those client extensions that have the optional ctl (component type list) attribute, and contain the id of the given component type or the reserved word “all”. If SE then lists all those client extensions which have the optional sel (server extension list) attribute and contain the id of the given server extension or the reserved word “all”.

Return arguments:
Returns the client extension elements.

Error values: Illegal argument. Insufficient role.

Required role: ModelUser.
Type: Obligatory.

Example:

```
<Request>
  <ListClientExtensions>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Type>CT</Type>
</ListClientExtensions>

<Return seq="1">
  <CE … />
  …
</Return>
```

3.1.31 ListClientExtensionBinaries

Gets the client extension binaries and the corresponding program ids.

Input arguments:

- Department: UUID of the Department.

Return arguments:

The client extension binaries and corresponding program ids. The return list contains no duplicates.

Error values: Illegal argument. Insufficient role.

Required role: Provider**.**

Type: Obligatory.

Example:

```
<ListClientExtensionBinaries>
  <Department>UUID<Department>
</ListClientExtensionBinaries>

<Return seq="1">
<CE bid=”…” />
…
</Return>
```

3.1.32 ListServerExtensionBinaries

Gets the server extension binaries and the corresponding program ids.

Input arguments:

- Department: UUID of the Department.

Return arguments:
The server extension binaries and corresponding program ids. The return list contains no duplicates.

Error values: Illegal argument. Insufficient role.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<ListServerExtensionBinaries>
  <Department>UUID<Department>
</ListServerExtensionBinaries>

<Return seq="1">
  <SE bid="…" pid="…"/>
  …
</Return>
```

3.1.33 ModiComponent

Modifies the component attributes.

Input arguments:

- Department: UUID of the Department.
- Id: UUID of the component.
- Name (opt): New value for name (i.e. renames component).
- Tracked (opt): New value for tracked attribute.

Return arguments:
None.

Error values: Illegal argument. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<ModiComponent>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Name>Amazing Pump</Name>
  <Tracked>true</Tracked>
</ModiComponent>

<Return seq=”1”/>
```

3.1.34 ModiEnumeration

Modifies the enumeration. The name cannot be changed, the existing allowed values cannot be renamed or removed. Only new values can be added. Implementation note: The rename and remove capabilities are restricted in order to ensure that the enumeration name used by component types and the value names used by components remain the same after creation. If this where not the case then the component types and components that have been copied out of the server would have inconsistent names.

Input arguments:

- Department: UUID of the department.
- Id: UUID of the enumeration.
- AllowedValue: New allowed value for the enumerator. Cannot contain blanks.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Allowed value already exists.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<ModiEnumeration>
  <Department>Department UUID</Department>
  <Id>UUID</Id>
  <AllowedValue>NewValue</AllowedValue>
  …
</ModiEnumeration>

<Return seq="1"/>
```

3.1.35 OpenComponentType

Sets the locked attribute of a component type to be false. After opening no modifications can be made to this component type.

Input arguments:

- Department: UUID of the Department.
- Id: Defines the id of Component Type.

Return arguments:
None.

Error values: Illegal id. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<OpenComponentType>
  <Department>UUID</Department>
  <Id>UUID</Id>
</OpenComponentType>

<Return seq=”1”/>
```

3.1.36 RemoveAccessRightsFromGroup

Removes access rights from a group in a department.

Input arguments:

- Department: UUID of the Department.
- Group: Name of the group.
- User: Name of the user whose access rights are removed.

Return arguments:
None.

Error values: Group does not exist. User does not exist. Access rights not in group. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<RemoveAccessRightsFromGroup>
  <Department>UUID</Department>
  <Group>Development Team</Group>
  <User>kernel</User>
</RemoveAccessRightsFromGroup>

<Return seq="1"/>
```

3.1.37 RemoveBinaryData

Removes binary data. Does not remove the references to this binary data! Intended to be used to remove corrupted data. Be careful out there!

Input arguments:

- Department: UUID of the Department.

- BinaryId: The id of the binary data.

Return arguments:
None.

Error values: Illegal argument. Insufficient role.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<RemoveBinaryData>
  <Department>UUID</Department>
  <BinaryId>…</BinaryId>
</RemoveBinaryData>

<Return seq="1">
```

3.1.38 RemoveCategory

Removes a category and its children. If the category contains open component types it cannot be removed. If the category contains enumeration elements which are referred to (i.e. have vector info elements that refer to it) it can not be removed.

Input arguments:

- Department: UUID of the Department.

- Id: Defines the id of category to remove.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

The server does not check if the category contains open component types or enumeration elements which are referred to.

Example:
```
<RemoveCategory>
  <Department>UUID</Department>
  <Id>UUID</Id>
</RemoveCategory>

<Return seq=”1”/>
```

3.1.39 RemoveClientExtension

Removes a client extension (including the corresponding binary data).

Input arguments:
- Department: UUID of the Department.

- Id: The client extension identifier.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

```
<RemoveClientExtension>
  <Department>UUID</Department>
  <Id>UUID</Id>
  </SetClientExtension>

<Return seq="1">
```

3.1.40 RemoveComponent

Removes a component.

Input arguments:

- Department: UUID of the Department.
- Component Id: The component identifier.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Consitency:

- If the component contains properties that are lifted the lifted properties must be deleted from the upper levels.

Example:

```
<RemoveComponent>
  <Department>UUID</Department>
  <ComponentId>UUID</ComponentId>
</RemoveComponent>

<Return seq="1"/>
```

3.1.41 RemoveComponentType

Removes a component type and its data. The component type can be removed only if it is locked.

Input arguments:

- Department: UUID of the Department.
- Id: Defines id of component type to remove.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider.
Type: Obligatory.

Example:
```
<RemoveComponentType>
  <Department>UUID</Department>
  <Id>UUID</Id>
</RemoveComponentType>

<Return seq=”1”/>
```

3.1.42 RemoveEnumeration

Removes the enumeration. The enumeration cannot be removed if any component type (in the type library in question) has vector info that refers to it.

Input arguments:
- Department: UUID of the department.

- Id: UUID of the enumeration.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Enumeration does not exist.

Required role: Provider.
Type: Obligatory.

```
<RemoveEnumeration>
  <Department>Department UUID</Department>
  <Id>Enumeration UUID</Id>
</RemoveEnumeration>

<Return seq="1"/>
```

3.1.43 RemoveGroup

Removes a group from a department.

Input arguments:
- Department: UUID of the Department.

- Group: Name of the group to remove.

Return arguments:
None.

Error values: Group does not exist. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:
```
<RemoveGroup>
  <Department>UUID</Department>
  <Group>Test Group</Group>
</RemoveGroup>

<Return seq="1"/>
```

3.1.44 RemoveLiftedProperty

Removes the lifted property.

Input arguments:
- Department: UUID of the department.

- Id: UUID of the lifted property.

Return arguments:
None.

Error values: Illegal argument, Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Consistency:

- If there is a chain of lifted properties (i.e. this lifted property has been lifted to an upper lever and so on) then all the upper lifted properties will also vanish.

- If this property was mapped (i.e. was pointed by an upper level mapping property) then the upper level mapping value will be set to null.

Release Notes for Gallery Setup 1.0:

Not implemented.

Example:

```
<RemoveLiftedProperty>
  <Department>UUID</Department>
  <Id>UUID</Id >
</RemoveLiftedProperty>

<Return seq="1"/>
```

3.1.45 RemoveProperty

Removes an optional property.

Input arguments:

- Department: UUID of the Department.
- Id: UUID of the Property.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<Request>
  <RemoveProperty>
  <Department>UUID</Department>
  <Id>UUID</Id>
  </RemoveProperty>

<Return seq="1"/>
```

3.1.46 RemoveRights

Removes the rights of a given element.

Input arguments:

- Department: UUID of the Department.
- Id: Defines the element the rights of which are removed.
- Type: Type of the parent element. Can be Dep, Ca, E, CE, SE, C, or CT.

- Group: Name of the group the rights of which you are removing.

- Recursive: If true then removes the rights of all child elements.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:
```
<RemoveRights>
  <Department>UUID</Department>
  <Id>5</Id>
  <Type>Ca</Type>
  <Group>Gallery Development Team</Group>
  <Recursive>true</Recursive>
</RemoveRights>

<Return seq="1"/>
```

3.1.47 RemoveServerExtension

Removes a given server extension. If the removed extension is the last reference to a binary then it will also uninstall the binary and remove it.

Input arguments:
- Department: UUID of the Department.

- Id: Defines the extension element.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role:
Provider.
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Does not check if the removed server extension is the last reference to a binary. Always removes the binary.

Example:

```
<RemoveServerExtension>
  <Department>UUID</Department>
  <Id>UUID</Id>
</RemoveServerExtension>

<Return seq="1"/>
```

3.1.48 RemoveUserFromGroup

Removes a user from a given group in a department.

Input arguments:

- Department: UUID of the Department.
- Group: Name of the group from which the user is removed.
- User: Name of the user to remove.

Return arguments:
None.

Error values: Group does not exist. User does not exist. User not in group. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

```
<RemoveUserFromGroup>
  <Department>UUID</Department>
  <Group>Development Team</Group>
  <User>kernel</User>
</RemoveUserFromGroup>

<Return seq="1"/>
```

3.1.49 RenameCategory

Renames a category.

Input arguments:

- Department: UUID of the Department.
- Id: Defines the id of the category to rename.

- Name: New name.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:
```
<RenameCategory>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Name>My Category</Name>
</RenameCategory>

<Return seq=”1”/>
```

3.1.50 RenameComponentType

Renames a component type. The locked attribute must be true.

Input arguments:
- Department: UUID of the Department.
- Id: Defines the id of the component type to rename.
- Name: New name.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

```
<RenameComponentType>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Name>My Category</Name>
</RenameComponentType>

<Return seq=”1”/>
```

3.1.51 SetAccessRightsToGroup

Adds an access rights element to a given group in a department.

Input arguments:

- Department: UUID of the Department.

- Group: Name of the group to which the user is added.

- User: Name of the user to add.

- Access: Access rights for the user. Can be none, read or write.

Return arguments:
None.

Error values: Group does not exist. User does not exist. Access rights already in group. Illegal name. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:
```
<SetAccessRightsToGroup>
  <Department>UUID</Department>
  <Group>Development Team</Group>
  <User>autksk</User>
  <Access>write</Access>
</SetAccessRightsToGroup>

<Return seq="1"/>
```

3.1.52 SetBinaryDataNew

Sets binary data. If the request contains the SetBinaryDataNew command, it cannot contain any other commands!

Input arguments:

- Department: UUID of the Department.

- Extension: Extension of the binary data id. The binary data id is made of two parts: <name>.<extension>. The server makes the name, but the client gives the extension.

- Binary data: The binary data is transferred after the request.

Return arguments:
Returns the binary data id.

Error values: Illegal argument. Insufficient role.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<Request>

<SetBinaryData>
  <Department>UUID</Department>
  <Extension>dll</Extension>
</SetBinaryData>
</Request>

\0\… (The binary data.)

<Response status="ok">
<Return seq="1">
  <BinaryId>666.dll</BinaryId>
</Return>
</Response>
```

3.1.53 SetBinaryDataOld

Sets old binary data. If the request contains the SetBinaryDataOld command, it can-not contain any other commands!

Input arguments:

- Department: UUID of the Department.

- Binary id: Id of the old binary data.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Data does not exist.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<Request>

<SetBinaryData>
  <Department>UUID</Department>
  <BinaryId>UUID</BinaryId>
</SetBinaryData>
</Request>


\0\?…

<Response status="ok">
    <Return seq="1"/>
</Response>
```

3.1.54 SetCategory

Sets the category.

Input arguments:

- Department: UUID of the Department.
- Parent: Defines the id of the parent category. If zero (0) then the parent is a department.
- Name: Name of the created category.
- Id: An UUID of the created element.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights. Name not unique. Ancestor is already type library.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<SetCategory>
  <Department>UUID</Department>
  <Parent>0</Parent>
  <Name>Pumps</Name>
  <Id>…</Id>
</SetCategory>

<Return seq=”1”/>
```

3.1.55 SetClientExtension

Sets the client extension. If the client extension exists then modifies it.

Input arguments:

- Department: UUID of the Department.
- ParentId: The parent category identifier.
- ClientExtension: The client extension data in XML format, must have name, type, binary id and id attributes. In addition it can have as many optional arguments as necessary.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role:
Provider.

Type: Obligatory.

```
<Request>
    <SetClientExtension>
        <Department>UUID</Department>
        <ParentId>…</ParentId>
        <ClientExtension[CE] name[n]=”…” type[t]=”…” binaryId[bid]=”…”
id=”…”  …/>
    </SetClientExtension>
</Request>
<Response status="ok">
    <Return seq="1">
</Response>
```

3.1.56 SetComponent

Sets a component to a category or a component. Must have write rights to parent and the used component types must exist and be open. The rights elements cannot be set with this command (they must be set with the modified component command) and are silently discarded. If the data contains a reference to a non-existent element it is set to null.

Input arguments:

- Department: UUID of the Department.

- ParentId: The parent identifier. The parent can be a category or a component.
- Type: Type of the element. Can be Ca, or C.
- Component: The component data in XML format, must be consistent with the component type data: all required properties must be given. The component data must include the component identifier (id attribute) and component type identifier (type id attribute). Each property and sub-component must also have a unique identifier.

Return arguments:
None.

Error values: Component exists already. Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<SetComponent>
  <Department>UUID</Department>
  <ParentId>UUID</ParentId>
  <Type>C</Type>
  <C id="…" n="my pump" tid="…">
    <C …>…</C>
     …
    <P id=”…” n=”my property”>
      <V>10</V>
    </P>
    …
  </C>
</SetComponent>

<Return seq="1"/>
```

3.1.57 SetComponentType

Sets the component type data. The component type data can be set only when the component type is locked (and there are no components in the component type). The operation checks that the input data is valid. (This includes, among other things, checking that the used enumeration types are valid.) The component type element must conform to the document type definition described in the get component type command. Each start/end terminal type must be defined in the terminal rules or it cannot be added.

Input arguments:

- Department: UUID of the Department.
- ParentId: The parent category identifier. (Parent must always be a category.)
- ComponentType: The component type data. Must contain an id.

Return arguments:
None.

Error values: Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:

```
<SetComponentType>
  <Department>UUID</Department>
  <ParentId>UUID</ParentId>
  <CT id=”…” n=”…” …>…</CT>
</SetComponentType>

<Return seq=”1”/>
```

3.1.58 SetDepartment

Creates a new department and sets the current user to its list of allowed users. Creates group “Everyone”. Creates a type library under the department.

Input arguments:

- Department: UUID of the new department.
- Name: Name of the new department.
- TypeLibraryId: UUID of the type library.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Department already exists.

Required role: KernelDeveloper**.**
Type: Obligatory.

Example:

```
<SetDepartment>
  <Department>UUID</Department>
  <Name>My Department</Name>
  <TypeLibraryId>UUID</TypeLibraryId>
</SetDepartment>

<Return seq="1"/>
```

3.1.59 SetDescription

Sets the description element.

Input arguments:

- Department: UUID of the Department.
- ParentId: UUID of the parent element.
- Type: Type of the element. Can be Dep, Ca, E, CE, SE, C, or CT.
- Description: Description text. If empty then the description element is removed.

Return arguments:

None.

Error values: Illegal argument. Insufficient role. Insufficient rights. Parent does not exist.

Required role: Provider**.**

Type: Obligatory.

Example:

```
<SetDescription>
  <Department>UUID<Department>
  <ParentId>UUID</ParentId>
  <Type>SE</Type>
  <Description>…</Description>
</SetDescription>

<Return seq="1"/>
```

3.1.60 SetEnumeration

Adds new enumeration or replaces data of the old one.

Input arguments:

- Department: UUID of the department.
- ParentId: UUID of the parent category (which must be a type library or a descendant of a type library).
- Enumeration: Enumerator data. Name of the enumeration must be unique among all enumerations in the containing type library. The allowed value cannot contain blanks.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Enumeration does not exist. Enumeration is used.

Required role: Provider**.**
Type: Obligatory.

```
<SetEnumeration>
  <Department>UUID</Department>
  <ParentId>UUID</ParentId>
  <E id="…" n="…">
    <AV>Value1</AV>
    …
  </E>
</SetEnumeration>

<Return seq="1"/>
```

3.1.61 SetGroup

Adds a group to a department.

Input arguments:

- Department: UUID of the Department.
- GroupName: Name of the group to add.

Return arguments:
None.

Error values: Illegal name. Insufficient role. Insufficient rights. Group already exists.

Required role: Provider**.**
Type: Obligatory.

Example:
```
<SetGroup>
  <Department>UUID</Department>
  <Group>Test Group</Group>
</SetGroup>

<Return seq="1"/>
```

3.1.62 SetLiftedProperty

Lifts a property to the parent component.

Input arguments:
- Department: UUID of the department.

- Parent: UUID of the component to which the lifted property will be added.

- Child: UUID of the lower level property that will be lifted.

- Id: The UUID of the new lifted property. The new lifted property will always point to the original property (i.e. its data value is the UUID of the lower level property and this cannot be changed by the SetReference command). If the original property is deleted the lifted property will also be deleted.

Return arguments:
None.

Error values: Illegal argument, Insufficient rights.

Required role:
ModelConfigurator.
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Not implemented.

Example:

```
<SetLiftedProperty>
  <Department>UUID</Department>
  <Parent>UUID</Parent>
  <Child>UUID</Child>
  <Id>UUID</Id >
</SetLiftedProperty>

<Return seq="1"/>
```

3.1.63 SetModel

Sets the component data.

Input arguments:
- Department: UUID of the Department.
- ParentId: UUID of the parent.
- Type: Type of the parent element. Can be Ca or C.
- Component data: The component data as it is returned by the GetModel command.

Return arguments:
None.

Error values: Component exists already.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:
```
<SetModel>
  <Department>UUID</Department>
  <ParentId>UUID</ParentId >
  <Type>Ca</Type>
  <C>…</C>
</SetModel>

<Return seq="1"/>
```

3.1.64 SetModified

Sets the modified element text.

Input arguments:
- Department: UUID of the department.
- Parent: The element UUID (can be category, component type, or component). If null then the department is the parent.
- Type: Type of the parent element. Can be Dep, Ca, E, CE, SE, C, or CT.

- At: The time of the modification. (There cannot be two modified elements at the same time.) The time format is YYYY-MM-DD HH:MM:SS.TTT (2002-05-13 14:28:21.426).

- Text: The text of the element.

Return arguments:
None.

Error values: Illegal argument, Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Example:

```
<SetModified>
  <Department>UUID</Department>
  <Parent>UUID</Parent>
  <At>2002-05-13 14:28:21.426</At>
  <Text>Descriptive text.</Text>
</SetModified>

<Return seq="1"/>
```

3.1.65 SetProperty

Sets a property value. If the property does not exist it is created, otherwise the data of the old property is replaced with the new data. Note that connections e.g. references from start terminal properties to end terminal properties cannot be set or removed with this function, use set reference instead. When creating mapping and terminal properties the property element data is quietly ignored. If the property is mapped or lifted then the original value is modified.

Input arguments:
- Department: UUID of the Department.

- Parent: UUID of the Component (used when the property does not exist).

- Id: UUID of the Property.

- Property: The property element in XML format. Must conform to the corresponding property info element and contain the property identifier. (See property and property info descriptions in the common element section.) If the property exists the content (value) is changed in which case the name attribute is optional.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

The server does not check the consistency of the property data with the corresponding property info. No support for lifted properties.

Example:
```
<SetProperty>
  <Department>UUID</Department>
  <Parent>Component UUID</Parent>
  <Id>Property UUID</Id>
  <P iid="…" …>
    <V>10</V>
  </P>
</SetProperty>

<Return seq="1"/>
```

3.1.66 SetReference

Connect a terminal, maps a terminal to a lower level or maps the mapping property to a lower level. Note that when making a connection (i.e. setting the first element of the start terminal property) the connection must obey the terminal rules. If a start terminal is connected then it cannot be mapped to the upper level (i.e. no second element of the upper level start terminal can point to a lower level start terminal which is connected), but the same is not true for an end terminal. When making a start terminal mapping (i.e. setting the second element of the startTerminal property) the pointed property must be the same type as the pointing property (and the pointed property cannot be connected). End terminal mapping can be made to any end terminal of the same type on the lower level (no matter if it is connected or not). The mapping properties can be mapped to any lower level mapping or normal properties.

Input arguments:
- Department: UUID of the Department.

- Id: UUID of the reference property and an optional vector element index if the property is not scalar. So for example if the UUID is a start terminal property and the index is one (1) this means you are making a connection and if the index is two you are making a mapping.

- Value: UUID of the end property and an optional index expression that defines the target element. An empty value is a null (void) pointer.

Return arguments:
None.

Error values: Illegal argument. Already mapped. Illegal connection. Type mismatch. Already connected.

Required role: ModelConfigurator**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Terminal rules are not checked. Optional index expression is not supported.

Example:

```
<SetReference>
  <Department>UUID</Department>
  <Id>UUID</Id >
  <Value>UUID</Value>
</SetReference>

<Return seq="1"/>
```

3.1.67 SetRights

Sets the rights for a given element. The element can be category, enumeration, component type, component, client extension, or server extension.

Input arguments:
- Department: UUID of the Department.

- Id: Defines the element the rights of which are set.

- Type: Type of the parent element. Can be Dep, Ca, E, CE, SE, C, or CT.

- Group: Name of the group the rights of which you set.

- Access: Defines access rights for the group. Can be none, read or write.

- Recursive: If true then all child elements get the same rights.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: ModelConfigurator**.**
Type: Obligatory.

```
<SetRights>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <Type>Ca</Type>
  <Group>Gallery Development Team</Group>
  <Access>write</Access>
  <Recursive>true</Recursive>
</SetRights>

<Return seq="1"/>
```

3.1.68 SetServerExtension

Sets the given extension. The server must check that the given binary id does not have any other program ids referring to it. (Each binary id can have only one unique server extension program id referring to it i.e. several server extensions can refer to one binary id, but they must all have the same program id.) Note that before you can set the server extension you must have set the server extension binary with the set binary data function (or selected an existing binary with list server extension binaries).

Input arguments:
- Department: UUID of the Department.
- ParentId: The parent category identifier.
- Id: UUID of the extension.
- Extension: The extension definition.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Release Notes for Gallery Setup 1.0:

The server does not check that the given binary id does not have any other program ids referring to it.

Example:

```
<SetServerExtension>
  <Department>UUID</Department>
  <ParentId>UUID<ParentId>
  <Id>UUID</Id>
  <SE n=”…” …>
  …
  </SE>
</SetServerExtension>

<Return seq="1"/>
```

3.1.69 SetSkeletonDepartment

Sets the department and its categories.

Input arguments:

- Output of GetSkeletonDepartment.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Insufficient rights.

Required role: KernelDeveloper**.**
Type: Obligatory.

Example:

```
<SetSkeletonDepartment>
  <Dep id=”…”…>
    </Ca …>…</Category>
  </Dep>
</SetSkeletonDepartment>

<Return seq=”1”/>
```

3.1.70 SetTerminalRules

Sets the terminal rules of a given type library.

Input arguments:

- Department: UUID of the Department.

- Id: UUID of the type library.

Return arguments:

None.

Error values: Illegal argument. Insufficient role.

Required role: Provider**.**

Type: Obligatory.

Example:

```
<SetTerminalRules>
  <Department>UUID</Department>
  <Id>UUID</Id>
  <TRs>
    <RR s=”flow”>
      <ESC n=”flow”/>
      …
    </RR>
    …
  </TRs>
</SetTerminalRules>

<Return seq="1"/>
```

3.1.71 SetTypeLibrary

Sets the type library.

Input arguments:

- Output of GetTypeLibrary.

Return arguments:

None.

Error values: Illegal argument. Insufficient role. Insufficient rights. Name not unique. Ancestor is already type library.

Required role: KernelDeveloper**.**

Type: Obligatory.

Release Notes for Gallery Setup 1.0:

Not implemented.

Example:

```
<SetTypeLibrary>
  <Department>UUID</Department>
    <Ca tl=”true” …>
    …
    </Ca>
  </Department>
</SetTypeLibrary>

<Return seq=”1”/>
```

3.1.72 SetUserDefinition

Adds a new user to the department.

Input arguments:

- Department: UUID of the department.
- Name: Name of the new user.

Return arguments:
None.

Error values: Illegal argument. Insufficient role. Department does not exist. User already exists. User must be activated by the system manager.

Required role: KernelDeveloper**.**
Type: Obligatory.

```
<SetUserDefinition>
  <Department>UUID</Department>
  <Name>…</Name>
</SetUserDefinition>

<Return seq="1"/>
```

3.1.73 SetUserToGroup

Adds a user to a given group in a department.

Input arguments:

- Department: UUID of the Department.
- Group: Name of the group to which the user is added.
- User: Name of the user to add.

Return arguments:
None.

Error values: Group does not exist. User does not exist. User already in group. Illegal name. Insufficient role. Insufficient rights.

Required role: Provider**.**
Type: Obligatory.

Example:
```
<SetUserToGroup>
  <Department>UUID</Department>
  <Group>Development Team</Group>
  <User>kernel</User>
</SetUserToGroup>

<Return seq="1"/>
```

4. Model Server Manager Data Model

4.1 Introduction

This chapter explains the Model Server Manager (MSM) data model. The MSM is a component providing location and launching services. This means that it keeps track of all the model servers configured into a host machine. It also provides services to launch a model server. The MSM data model and query language describe the logical data model and the functional interface to MSM services.

4.2 MSM

Description:

The required root element of the XML document. Note that ModelDeveloper role used in the example refers to the ModelConfigurator role used elsewhere in the document.

Example:

```
<MSM>
<Port>7777</Port>
<User name="VTT kernel developer" role="KernelDeveloper" />
<User name="VTT provider" role="Provider" />
<User name="VTT model developer" role="ModelDeveloper" />
<User name="VTT model user" role="ModelUser" />
…
<ModelServer name="Gallery">
  <Class>Database</Class>
  <Address>130.188.18.136</Address>
  <Port>0</Port>
  <Module>F:\Galleria\bind\gs.exe</Module>
  <Database>F:\Galleria\db\gallery</Database>
  <StartDirectory>F:\Galleria\bind</StartDirectory>
  <LaunchUser>GMS</LaunchUser>
  <LaunchPassword>NLCKFGEJIFJCGBHN</LaunchPassword>
  <OPCProtocol>HTTPS</OPCProtocol>
  <OPCProgId>VTT.OPCXMLProxy.1</OPCProgId>
  <OPCTarget>ESPT120057/ProsimProxyOPCXMLStub</OPCTarget>
  <User name="VTT kernel developer" role="KernelDeveloper"/>
  <User name="VTT provider" role="Provider"/>
  <User name="VTT model developer" role="ModelDeveloper"/>
  <User name="VTT model user" role="ModelUser"/>
  …
</ModelServer>
…
</MSM>
```

4.3 Port of the MSM

Description:
This is the port number of the model server manager. All clients must use this port when requesting services.

Example:

```
<Port>7777</Port>
```

4.4 MSM User

Attributes:
- Name: The name of the user.

- Role: The role of the user. Can be KernelDeveloper, Provider, ModelConfigurator (ModelDeveloper in implementation), or ModelUser.

Description:
Defines the model server manager user. Only the users of the MSM can access the services of this MSM.

Example:

```
<User name="VTT kernel developer" role="KernelDeveloper" />
```

4.5 Model server

Attributes:
- Name: The name of the model server.

Child elements: Class, Address, Port, Module, Database, StartDirectory, LaunchUser, OPCProtocol, OPCProgId, OPCTarget, and User.

Description:
This element groups all the attributes of the model server together.

Example:

```
<ModelServer name="Gallery">
  <Class>Database</Class>
  <Address>www.simulationsite.com</Address>
  <Port>0</Port>
  <Module>F:\Galleria\bin\gs.exe</Module>
  <Database>F:\Galleria\db\gallery</Database>
  <StartDirectory>F:\Galleria\bin</StartDirectory>
  <LaunchUser>GMS</LaunchUser>
  <LaunchPassword>NLCKFGEJIFJCGBHN</LaunchPassword>
  <OPCProtocol>HTTPS</OPCProtocol>
  <OPCProgId>VTT.OPCXMLProxy.1</OPCProgId>
  <OPCTarget>ESPT120057/ProsimProxyOPCXMLStub</OPCTarget>
  <User name="VTT kernel developer" role="KernelDeveloper"/>
  <User name="VTT provider" role="Provider"/>
  <User name="VTT model developer" role="ModelDeveloper"/>
  <User name="VTT model user" role="ModelUser"/>
  …
</ModelServer>
```

4.6 Class of the Model Server

Description:
This is the class of the model server. Can be Database or Simulator. A simulator server should have all the functionality of a Database model server and in addition it should support the functionality for simulation of models.

Example:

```
<Class>Database</Class>
```

4.7 Address of the Model Server

Description:
This is the Internet address of the model server. This is the address clients use when connecting to this service.

Example:

```
<Address>www.simulationsite.com</Address>
```

4.8 Port of the Model Server

Description:
This is the port number of the model server manager. All clients must use this port when requesting services of the model server in question.

Example:

```
<Port>0</Port>
```

4.9 Executable of the Model Server

Description:
This is the main executable of the model server. This executable is started when this model server is launched.

Example:

```
<Module>F:\Galleria\bin\gs.exe</Module>
```

4.10 Database directory of the Model Server

Description:
This is the database directory of the model server. This directory contains all the files that form the database of the model server in question.

Example:

```
<Database>F:\Galleria\db\gallery</Database>
```

4.11 Start directory of the Model Server

Description:
This is the work directory of the launching process.

Example:

```
<StartDirectory>F:\Galleria\bin</StartDirectory>
```

4.12 Launching User

Description:
This user owns the launched process.

Example:

```
<LaunchUser>GMS</LaunchUser>
```

4.13 Launching Password

Description:
This is the crypted password for the launching user.

Example:

```
<LaunchPassword>NLCKFGEJIFJCGBHN</LaunchPassword>
```

4.14 OPC Protocol

Description:
This is the used communication mechanism in the data transmission. Can be COM, DCOM or HTTPS.

Example:
```
<OPCProtocol>HTTPS</OPCProtocol>
```

4.15 OPC Programmatic Id

Description:
The COM program id for the OPC server. All clients must use this when requesting services.

Example:

```
<OPCProgId>VTT.OPCXMLProxy.1</OPCProgId>
```

4.16 OPC Target

Description:
If the protocol is HTTPS then this field indicates the target for the communication.

Example:

```
<OPCTarget>www.simulationsite.com/ProsimProxyOPCXMLStub</OPCTarget>
```

4.17 User of the Model Server

Description:
This is one of the allowed users for this Model Server. The format is the same as with the user element of the model server manager.

Example:

```
<User name="VTT kernel developer" role="KernelDeveloper"/>
```

5. Model Server Manager Query Language

5.1.1 Introduction to query functions

The model server manager is a component that provides location and launching services to the model clients.

The client sends its request and receives the response with the mechanism provided by the transport protocol. The request is not a complete XML document because it does not use the XML header (<?xml version=”1.0” encoding=”ISO-8859-1”?>). At the moment the version must always be 1.0 and the encoding must always be ISO-8859-1. Thus they do not need to be transferred by each request. The request contains an element for each requested service, for example:

```
<Request number=”1” user=”…” password=”…”>
    …
    <GetDom/>
    …
</Request>
```

Each request is given a response that contains a response element for each (first level) element in the request. The response element contains the same number element as the request. If the service call has failed the return element for that service is ”<Error>”. All the service calls are separate transactions, so other clients can change the state of the server between function calls. (Because there are no services for configurating the MSM configuration the only thing that normally changes is the port number. The configuration can change if someone stops and starts MSM). If one function call fails the following function calls might be discarded.

In the following function descriptions the function name is the header (e.g. SetPort) and the input aguments are described first (e.g. ModelServer). The input arguments are transferred to attributes of the corresponding function element (e.g. <SetPort modelServer=”…” …). The return arguments are described after the input arguments.

Note that in addition to being among the users listed in the MSM database you are authenticated by the operating system user account and password. Even though the attribute names in the following descriptions start with a capital letter, in the XML elements they always start with a lowercase letter (e.g. <SetPort modelServer=”…” …/>).

5.1.2 GetDom

Gets the MSM database.

Input arguments:
None.

Return arguments:
- Model server manager database in XML format.

Error values: Failed to get DOM.

Example:

```
<Request number=”1” user=”…” password=”…”>
    <GetDom/>
</Request>
<Response number="1">
    <MSM>…</MSM>
</Response>
```

5.1.3 Authenticate

Authenticates that the user has a valid user id to the operating system with the correct password. Also authenticates that the user is also a user of the given model server.

Input arguments:
None.

Return arguments:
- Model server manager database in XML format.

Error values: Failed to authenticate.

Example:

```
<Request number=”1” user=”…” password=”…”>
    <Authenticate modelServer=”…” user=”…” password=”…”/>
</Request>
<Response number="1">
    <MSM>…</MSM>
</Response>
```

5.1.4 SetPort

Sets the model server port to a given value. The user must be the launching user for this command to succeed.

Input arguments:

- ModelServer: Name of the model server.
- Port: Number of the port of the given model server.

Return arguments:

Model server manager database in XML format.

Error values: Failed to get DOM.

Example:

```
<Request number=”1” user=”…” password=”…”>
    <SetPort modelServer=”name” port=”666”/>
</Request>
<Response number="1">
    <MSM>…</MSM>
</Response>
```

5.1.5 Start

Starts the model server.

Input arguments:

- ModelServer: Name of the model server.

Return arguments:

- Model server manager database in XML format.

Error values: Server not found. Server already running. Port not zero. Could not start server. Could not start server (2). Failed to get DOM.

Example:

```
<Request number=”1” user=”…” password=”…”>
    <Start modelServer=”name”/>
</Request>
<Response number="1">
    <MSM>…</MSM>
</Response>
```

Acknowledgements

Gallery project was funded by the National Technology Agency (Tekes), CTS Engineering, Fortum Nuclear Services, GL&V Finland, Metso, m-Real, Sulzer Pumps Finland and VTT Industrial Systems. During almost three years 14 research scientists and trainees from VTT Industrial Systems, Helsinki University of Technology and Lappeenranta University of Technology have participated to the project. This document is one of the end products of the work done in the project.

References

Gallery, 2002. Gallery web site, http://www.simulationsite.com/gallery [referenced 27.11.2002].

Karhela, T. 2002. A Software Architecture for Configuration and Usage of Process Simulation Models Software component technology and XML based approach. Teoksessa: VTT Information Service. VTT Publications 479. 129 p. + app. 19 p. ISBN 951-38-6012-4

Published by	Series title, number and report code of publication VTT Research Notes 2184 VTT–TIED–2184

Author(s)
Kondelin, Kalle & Karhela, Tommi

Title

Gallery Markup and Query Language Specification

Abstract

Gallery is a software infrastructure for storing, deploying and using process model and parameter data. This document is compiled for Gallery model server and client developers as well as for Gallery client and server extension developers. It describes the data model (GML) and main interfaces (GQL) of the Gallery system. Gallery was developed in a research project (2000–2002) funded by National Technology Agency (Tekes) and Finnish industry. This document is one of the end products of the project.

Keywords
Gallery Markup Language, Gallery Query Language, software infrastructure

Activity unit
VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland

ISBN 951–38–6117–1 (soft back ed.) 951–38–6118–X (URL: http://www.inf.vtt.fi/pdf/)			Project number
Date January 2003	Language English	Pages 111 p.	Price C
Name of project Gallery project		Commissioned by National Technology Agency (Tekes), CTS Engineering, Fortum Nuclear Services, GL&V Finland, Metso, m-Real, Sulzer Pumps Finland, VTT Industrial Systems	
Series title and ISSN VTT Tiedotteita – Research Notes 1235–0605 (soft back edition) 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU	VTT INFORMATIONSTJÄNST	VTT INFORMATION SERVICE
PL 2000	PB 2000	P.O.Box 2000
02044 VTT	02044 VTT	FIN–02044 VTT, Finland
Puh. (09) 456 4404	Tel. (09) 456 4404	Phone internat. + 358 9 456 4404
Faksi (09) 456 4374	Fax (09) 456 4374	Fax + 358 9 456 4374

ISBN 951–38–6117–1 (soft back ed.) ISBN 951–38–6118–X (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1235–0605 (soft back ed.) ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)

RESEARCH NOTES 2184 Gallery Markup and Query Language Specification