1. Purpose of actions

Actions are used to create XML-files. Action language, RuleX, is used as an instruction to define what parts and what content of XML is to be modified or created.

Creation of new XML files is done by using an existing, before-made XML file as a basis. This target XML file where the actions are executed is called *template file*. Template file is not modified by actions, instead, new files are made based on template file and the defined actions. In myXML, template file is to be separately defined.

Like validation rules, actions language is written against a schema and may use simple and complexTypes as a context for the action. OCL expressions are used for defining conditions and queries within action.

The nature of creating and editing XML files requires specialised language and methods. The purpose of this wiki page is to explain syntax of RuleX and available methods, as well as to provide examples of useful actions.

2. Practices used in this action guide

2.1. Schema

Unless stated otherwise, the examples provided in this used guide use the same schema as the OCL user guide. For reference, attached is a screenshot of the schema in XMLspy and below is an image how the schema is seen in myXML.

Schema element	Type	Occ.
Message	Message	1...1
Header	HeaderType1	1...1
Id	Max35Text	1...1
TimeStamp	DateTime	1...1
ControlSum	decimal	1...1
NumberOfTransactions	integer	0...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Transaction	TransactionType1	1...*
Id	Max35Text	1...1
Amount	decimal	1...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Creditor	PartyIdentification	1...1
Name	Max140Text	0...1

2.2. Template file

The XML-file example actions are modifying use the attached template file, unless otherwise stated.

2.3. Code-blocks and ensuing XML

Example actions are always provided in a code block as shown below. Context of the action is stated before

the code block. Context of the below example is *TransactionType1.*

```
// this line contains a comment. next line contains action stuff
if(self.Debtor->size() = 1) then
  self.Debtor : "New name";
endif;
```

The XML example code generates is surrounded with block quotes. Only the relevant part of the XML is shown. Note that actions may modify other parts of XML as well, which is not shown in the example due to space reasons.

```
<Transaction>
  <Id>TxId1</Id>
  <Amount>2</Amount>
  <Debtor>
    <Name>New name</Name>
  </Debtor>
  <Creditor>
    <Name>Creditor1</Name>
  </Creditor>
</Transaction>
```

Certain parts of the action and XML may be highlighted to draw attention to important parts of the example.

2.4. Terminology

Below terminology is used in this guide and is relevant when writing actions.

Term	Definition
Statement	As opposed to a sentence in natural language. Complete unit of execution. Terminated by semicolon ;. Usually one row in expression
Expression	Construct made with variables, operators, and method invocations. Action text box in myXML. May contain one or more statements
Left-hand-side	Left side of assign operator :
Right-hand-side	Right side of assign operator :

Term	Definition
Primitives	Int, double, string, etc. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. In RuleX, keyword auto may be used instead of explicitly defining primitive type
Node	Item within tree, containing value or links to children nodes. In XML, either complexType or simpleType element
Tree	Collection of nodes, starting from root. Can be though of as XML-schema (.xsd)
Root	First node in a tree
Branch	Node with children. complexType in XML
Leaf	Node without children. simpleType in XML
complexType	XML type containing child elements. Does not contain value
simpleType	XML type contaning value. Does not contain links to further elements. Therefore a leaf-node
Variable assign	Assignment of a variable in left-hand-side with a (new) value
Direct assign	Assigment of a node within tree or within varuable

3. Syntax of RuleX - properties of the language

This section contains example RuleX statements. More thorough examples can be seen in section 5

Comments

Inline comments are prefixed by //, multiline comments are surrounded by /** and **/.

Statements

Statements are completed with semicolon, character ;

Set-operation

Operand : states a set operation. Set means to assign content of left-hand-side of statement with right-hand-side. Content to be set may either be a variable or XML directly.

Below is an example of setting a variable

string someVar : "text";

Below is an example of direct assign

self.Cdtr.PstlAdr.Ctry : "NO";

If-then-else -statement

If - then - else statement is used by giving a boolean OCL expression within the brackets of if(). Statement has to contain "then" and end with "endif;", else is optional.

```
if (self.Amount > 200 and self.Creditor.Name->size() = 1) then
  self.Creditor.Name : "Major receiver";
else
  self.Creditor.Name : "Private receiver";
endif;
```

Tree traversal

Navigation of the schema tree is done similarly as in OCL, "self" always refers to the context and dot notation is used to traverse the elements within the tree, from parent to child elements and attributes.

Generating data

When generating data with data-generation-methods, it can be assumed that the generated content will be valid according to schema. Generating "bad-data" has its own methods, where the fact is stated in the method name.

The following methods will always generate schema-valid data:

- createFull()
- createMandatory()

However, please note that it is possible to create schema invalid content even when not using methods specialised for this purpose. For example it is possible to remove a mandatory element with delete() which would then ensue in schema invalid XML-file

Assigning data

When assigning content existing data is replaced. Additionally, if element to be setted does not exist but its parent element does exist, new node is created with the defined value. New node is not created if the parent does not exist. Therefore, it is important to note that assignment may not be successfull, depending on the content currently present in template file / variable.

In other words, assignment has three options depending on template file. Option 1 and 2 produce the same outcome to the output file

1. Existing value in simpleType is replaced
2. simpleType element is created, value is assigned and simpleType element is assigned to its parent

3. If parent of simpleType does not exist, nothing is done

Let's assume we have the following content in a template file

```
<Creditor>
   <Name>Creditor1</Name>
</Creditor>
```

Therefore, in a direct assignment case below

self.Creditor.Name : "Bowien daavidi";

1) Existing value of element 'Name' will be replaced with the value defined in right-hand-side. The output file will contain elements 'Creditor / Name' with value "Bowien daavidi".

If the template file does not contain name

```
<Creditor>
</Creditor>
```

2) New node is created as the parent exists. Name is created and assigned with the value and Creditor is assigned with the newly created Name. Therefore this output file as well will contain elements 'Creditor / Name' with value "Bowien daavidi".

3) However, if the template file does not contain Creditor, action does not create the complexType element Creditor and therefore the output file will not contain the value defined in the statement. It can be thought in a way that in this case "Name" is created with a value, but it cannot be assigned anywhere as "Creditor" does not exist. The output file will be the same as the input file.

Stating the type of element in methods

Type has to be explicitly stated in cases whenever it is not deductable from the left-hand side of statement.

Therefore, it is not required to state the type in argument of createNode when directly assigning an element, for example in

self.Creditor : createNode();

In the statement above, we know that we are dealing with element "Creditor" and thus we already know it's type, it doesn't have to be stated again.

However, when we are creating a variable we cannot determine the type from the left-hand side. In this case type is required to be stated

auto typeVar : createNode(PartyIdentification);

The outcome here is that both of the examples created a type "PartyIdentification". The upmost example assigned the type already to XML, for Creditor. The downmost example assigned the type to a variable which may then be used later. The downmost example did not modify any XML itself.

Stating the type of variable and the keyword "auto"

When we are creating a variable, we may explicitly define the variable in the left-hand-side:

string varString1 : "teksti1";

however, keyword auto is supported as well, which determines the type by right-hand-side:

auto varString2 : "teksti2";

This is supported for simple and complexTypes as well

auto typeVar1 : createNode(PartyIdentification); // with auto
PartyIdentification typeVar2 : createNode(PartyIdentification); // explicitly defining the variable

When auto is used, the type again is determined from the right-hand-side.

The goal of the near-future development of RuleX is to offer additional ways for the writer to locate the correct type, without the need to remember or copy the long simple/complexTypes defined by schemas. Auto keyword was developed specifically this in mind.

4. Available operations and methods

Table below lists available methods and their syntax in actions language. Further description and examples are provided in next sections.

Operation / Method	Syntax	Definition
Set operation	variable1 : "value"; variable2 : variable1; element : "value"; element : variable1;	Sets left-hand-side of the statement (XML or variable) with value in right-hand-side. Note that in case of simpleType, content will also be created if it does not already

Operation / Method	Syntax	Definition
		exists
createNode	auto variable : createNode(type); self.Header : createNode();	Creates a node. Node is either complex or simpleType in XML. Existing node is replaced. May be used for a variable or directly for element. When used for variable, type of element has to be explicitly defined in an argument
createFull	auto variable : createFull(type); self.Header : createFull();	Creates full information to given type while maintaining schema-validity. This includes elements as well as valid dummy-data. When choice-structure is present, creates first choice. When multiplicity elements are present, creates one occurrence May be used for direct assignment or for variable assignment. When used for simpleType, creates dummy-data
CreateMandatory	auto variable : createMandatory(type); element : createMandatory();	Creates all mandatory information to given type while maintaining schema-validity. This includes elements as well as valid dummy-data. Does not generate optional elements. If choice-structure is defined with both choices having multiplicity of 1..1, creates first choice. If collection elements are defined, creates at least 1 element. When mandatory elements contain mandatory children, those are generated as well. May not be used for simpleTypes. Otherwise same syntax as with createFull()

Operation / Method	Syntax	Definition

createBranch	auto varBrnch : createBranch(HeaderType1, Id, "Val");
	
	self.Header : createBranch(Id, "val2");
		Creates branch element or elements, depending on the given path-argument.
		
		Replaces existing content in a branch.
		
		Last element defined in the left-hand-side is replaced completely with the branch-elements defined in this method. Therefore this method efficiently wipes out any possible existing content in a branch.
		
		Items created may be complex types and simpleType. Supports assigning a value to the leaf of a branch, this last argument is optional.
		
		May be used for direct assign as well as for variable assign.
		
		When assigning XML directly, first argument determines the path/ elements created
		
		When assigning on variable, first argument defines the type of first element in the branch and second argument determines the further path to be generated

Copy	auto varTx : copy(node);	Copies a type. Argument inside brackets expects an ocl query or existing node
Delete	element.delete();	Deletes element. Expects no arguments. May not be used for collection
Add	parentElement.child.add(childvariable);	Adds an element into a collection May only be used for collection elements. Newly added element is added as the

Operation / Method	Syntax	Definition
		last item

If Then	if (ocl invariant) then
	  Action;
	else
	  Action;
	endif;
		if-else-statement. Expects a boolean from an OCL expression

Foreach	foreach varCollectionItem in (collection) varCollectionItem : "newValue"; endeach;	Looping thorough a collection. Variable after word "foreach" defines the individual item within the collection. Collection defined within brackets after keyword "in", excpets an OCL query Always finished with endeach;
save()	save(varType, "Msg_" + i.toString() + ".xml");	Saves the type indicated in parameter

Method argument usage

Arguments are method specific and accepted argument amount depends on wheter they are used in direct assignment or whether var is assigned.

Methods requiring type as an argument accept the type with or without quotation marks. However, quotation marks are required if the type contains certain RuleX specific characters.

The argument usage for each method can be seen from below tables.

- Text indicates argument to be mandatory
- Optionality is indicated with brackets
- non-applicability is indicated with dash.

createNode()

	Argument 1	Argument 2

Direct assign	(value)	-
Local var assign	type	(value)

createFull()

	Argument 1	Argument 2
Direct assign	-	-
Local var assign	type	-

createMandatory()

	Argument 1	Argument 2
Direct assign	-	-
Local var assign	type	-

createBranch()

	Argument 1	Argument 2	Argument 3
Direct assign	path	(value)	(value)
Local var assign	type	path	(value)

copy()

	Argument 1	Argument 2
Direct assign	node	-
Local var assign	node	-

delete()

	Argument 1	Argument 2

Direct assign	-	-
Local var assign	-	-

add()

	Argument 1	Argument 2
Direct assign	node	-
Local var assign	node	-

save()

	Argument 1	Argument 2
Usage	type	filename

5. Description of available operations and methods

This sections describe the usage of action operations and methods in further detail and provides usage examples.

5.1. Set action

Sets the value of an XML element or a variable. The assignment operator is colon :

[element/variable] : value;

Syntax of *Set* is the same as syntax of *Create & Set*. This means that depending on content in template file, elements may also be created unless otherwise defined in action language. Further details in section Create & Set.

5.1.1. Examples of Set

In these examples, context is *TransactionType1* and template file is as defined in section 2.2. As the type *TransactionType1* exists multiple times in the template file, all these types are modified with actions. Only one Transaction block is provided in the resulting example XML here.

Set Example 1: Setting a string element

```
self.Creditor.Name : "Mad Max";
```

Resulting XML

```
<Transaction>
  <Id>TxId3</Id>
  <Amount>2</Amount>
  <Creditor>
    <Name>Mad Max</Name>
  </Creditor>
</Transaction>
```

Set Example 2: Setting an integer element

```
self.Amount : 300;
```

Resulting XML

```
<Transaction>
  <Id>TxId3</Id>
  <Amount>300</Amount>
  <Creditor>
    <Name>Creditor3</Name>
  </Creditor>
</Transaction>
```

Set Example 3: Setting a simpleType element within a newly created complexType

When setting a complexType element, the type must already exist and be of the same type where it is setted. First line of this example includes creating complexType, described in the following section.

```
auto varPartyId : createNode(PartyIdentification);
varPartyId.Name : "Heisenberg";
self.Debtor : varPartyId;
```

Resulting XML

```
<Transaction>
  <Id>TxId3</Id>
  <Amount>2</Amount>
  <Debtor>
    <Name>Heisenberg</Name>
  </Debtor>
```

```
    <Creditor>
      <Name>Creditor3</Name>
    </Creditor>
  </Transaction>
```

The above example creates Debtor if it doesn't already exist in template file and sets its name to the defined value. If Debtor and Name aleady exist the name is modified.

5.2. CreateNode

Creates an XML element. Keyword *auto* is used to define variables in a statement.

Created type type can be simpleType or complexType.

Using an optional second parameter in create sets sets the value for the element.

auto varName : create(type, (value));

5.2.1. Examples of createNode

In these examples, context is *TransactionType1* and template file is as defined in section 2.2.

Create Example 1A: Creating string and int. Separately assigning values and finally modifying XML

```
// Explicitly stating created variable types
auto varExample1 : create(string);
auto varExample2 : create(int);

// Assigning values
varExample1 : "text value";
varExample2 : 600;

// Modifying XML
self.Id : varExample1;
self.Amount : varExample2;
```

Create Example 1B: Creating and setting string and int in one statement

In order to simplify example 1a, we can combine create with set. Type is defined based on right-hand side of auto-statement.

```
// Creating variables and assigning values
auto varExample1 : "text value";
auto varExample2 : 600;
```

```
// Modifying XML
self.Id : varExample1;
self.Amount : varExample2;
```

Create Example 1C: Creating and setting string and int in one statement

We can simplify the example further, by creating, assigning and modifying in one statement. This example is the same one as Example 1 in set.

```
// Modifying XML with below values
self.Id : "text value";
self.Amount : 600;
```

If Id or Amount would not exist in template file they would be added to the resulting output file.

Resulting XML for 1A, 1B and 1C

```
<Transaction>
  <Id>text value</Id>
  <Amount>600</Amount>
  <Creditor>
    <Name>Creditor2</Name>
  </Creditor>
</Transaction>
```

Create Example 2: Creating and setting simpletype under complexType

When creating and setting, simpleTypes are created and added to ensuing output XML. However, complexTypes are not. In this example, the simpleType to be set is a child of a complexType.

self.Debtor.Name : "Devin Townsend";

Template file has three occurrences of Transaction, one where Debtor/Name is given, one where Debtor exists without name and one without Debtor. This ensues in situation where first occurrence of Debtor/Name is modified, second occurrence of Debtor has Name created and modified and third occurrence is untouched, as the complexType Debtor does not exist.

Non-relevant content omitted from below output

```
<Transaction>
    <Debtor>
```

```
            <Name>Devin Townsend</Name>
        </Debtor>
    ....
</Transaction>
<Transaction>
    ...
    <Debtor>
            <Name>Devin Townsend</Name>
    </Debtor>
    ...
</Transaction>
<Transaction>
    ...
</Transaction>
```

Set example 3 has a similar action, with the difference that in *Set example 3* complexType was individually created and the complexType was explicitly added to Debtor. Difference in the output file is that in *Set Example 3* every occurrence of Transaction will contain Debtor/Name with the defined value.

Create example 3: Creating and setting nested elements

ComplexTypes are created in a same manner as schema-built-in types. The name of the type is given as a parameter of create. The type of complexType created must exist in the schema.

In this exampe contex of the action is *Message.*

```
// Creating complexTypes
auto varHeader : createNode("HeaderType1");
auto varPartyId : createNode("PartyIdentification");

// Creating and assigning simpleType
varPartyId.Name : "created name";

// Creating and assigning complexType Header/Debtor to be the previously created
complexType
varHeader.Debtor : varPartyId;

// Setting the header in the template file to be the newly created header
self.Header : varHeader;
```

```
<Header>
    <Debtor>
        <Name>created name</Name>
    </Debtor>
</Header>
```

This output file is invalid according to schema as mandatory elements are missing.

Create example 4: Create with additional parameter

This example produces an identical output as *create example 1*.

```
// Explicitly stating created variable types. Using second parameter to set the value
auto varExample1 : createNode(Max35Text, "Text value");
self.Id : varExample1;
```

```
<Id>Text value</Id>
```

5.3. Generating data

Generating dummy-data is possible for following types:

- date
- dateTime
- string
- int

The methods available for each type is described in the examples below.

5.3.1. Examples for date and dateTime

Datetime and date elements can be created based on today's date, and dates relative to today's date. The W3 formats for date and dateTime are used: CCYY-MM-DD and [-]CCYY-MM-DDThh:mm:ss[Z|(+|-)hh:mm]. Where different date formats are required, the created dates can be reformatted using string manipulation functions.

Context for all these examples is *HeaderType1*.

Generating data, example 1: creates a dateTime equal to the current timestamp.

```
self.TimeStamp : dateTime.today();
```

```
<TimeStamp>2016-09-12T12:34:38</TimeStamp>
```

DateTime methods exist for date as well

self.TimeStamp : date.today();

This would work if the type of TimeStamp would be date. The output would be in type *date* with format of "2015-02-24"

Generating data, example 2: Creating dateTime element with value relative to today's date.

The parameter of today takes an int value marking the offset for the day. Negate values mark the date to be in past.

self.TimeStamp : dateTime.today(10);

<TimeStamp>2016-09-22T12:50:29</TimeStamp>

Generating data, example 3: Creating date element with integer format CCYYMMDD and string format CCYY-MM-DD

```
auto varDateInt : date.today().toString().replaceAll("-","").toInteger();
auto varDateStr : date.today().toString();
self.NumberOfTransactions : varDateInt;
self.Id : varDateStr;
```

```
<Header>
   <Id>2016-09-12</Id>
   <TimeStamp>2015-07-03T12:17:50</TimeStamp>
   <ControlSum>6</ControlSum>
   <NumberOfTransactions>20160912</NumberOfTransactions>
</Header>
```

5.3.2. Examples of generating strings and integers

Context for all these examples is HeaderType1.

Generating data, 4: Generating unique and random values

For certain fields, unique values are needed. For that purpose String.shortUniqueTimeStamp is most suitable. It uses current timestamp + random number to generate unique value on each execution and the output contains both characters and digits. Example below:

```
auto uniquestr : String.shortUniqueTimeStamp();    //generates e.g. "JtDOC"
self.Id : uniquestr;
```

Generating data, 5: Generating random strings and ints

It is also possible to generate random values with chars or digits only. The length of generated value is given as a parametr to these .random() methods.

```
self.Id : string.random(10);                    //generates e.g. <Id>COFBNrkvkA</Id>
self.NumberOfTransactions : int.random(5);       // generates e.g.
<NumberOfTransactions>98757</NumberOfTransactions>
```

5.4. CreateFull action

The method createFull generates the full content available for the type given in a parameter.

auto varType : createFull("type");

Data is created based on information available in the schema. Created dummy-data is valid according to schema

5.4.1. Examples of createFull

Below expression creates full data available for type *PartyIdentification* and sets the generated data to *Debtor*.

```
auto varParty : createFull("PartyIdentification");
self.Debtor : varParty;
```

```
<Transaction>
   <Id>a</Id>
   <Amount>2</Amount>
   <Debtor>
      <Name>aaaaaaaaaaaaaaaaa</Name>
   </Debtor>
   <Creditor>
```

```
        <Name>Creditor1</Name>
    </Creditor>
</Transaction>
```

5.5. Copy action

Copies a complexType XML element. Copied element is assigned in the parameter.

auto varCopyOfElement : copy(element);

5.5.1. Examples of copy

Example 1: Copying a complexType element

```
auto varCopyOfCreditor: copy(self.Creditor);
self.Debtor : varCopyOfCreditor;
```

```
<Transaction>
    <Id>a</Id>
    <Amount>2</Amount>
    <Debtor>
        <Name>Creditor1</Name>
    </Debtor>
    <Creditor>
        <Name>Creditor1</Name>
    </Creditor>
</Transaction>
```

5.6. Delete action

Deletes an XML element. Used by using dot notation after the element to be deleted. Delete() takes no parameters.

element.delete();

Note that delete() does not work for collections and instead, each individual item within collection has to

be treated separately. To see an example on how to do this, please view foreach section of the guide.

5.6.1. Examples of delete

Delete example 1: Deleting a child element

```
self.Debtor.delete();
```

```
<Transaction>
   <Id>Tx1</Id>
   <Amount>2</Amount>
   <Creditor>
      <Name>Creditor1</Name>
   </Creditor>
</Transaction>
```

Delete example 2: Deleting a grandchild element

```
self.Debtor.Name.delete();
```

```
<Transaction>
   <Id>Tx1</Id>
   <Amount>2</Amount>
   <Debtor/>
   <Creditor>
      <Name>Creditor1</Name>
   </Creditor>
</Transaction>
```

5.7. Add action

Adds an XML child element to a parent element. The element being added must already exist.

Add is used in the case where a parent element can contain more than one instances of the same child element (maxOccurs is greater than 1 for the child element). If maxOccurs is 1 set action may be used.

parentElement.add(childElement);

5.7.1. Examples of add

Examples use the context *Message*

Example 1: Adding an element

Transactions are added to the end of the file.

```
// Creating transaction, assigning values
auto tx1 : createNode("TransactionType1");
tx1.Id : "1";

// Creating transaction, assigning values
auto tx2 : createNode("TransactionType1");
tx2.Id : "2";

// Adding transactions to the file
self.Transaction.add(tx1);
self.Transaction.add(tx2);
```

```
<Transaction>
   <Id>1</Id>
</Transaction>
<Transaction>
   <Id>2</Id>
</Transaction>
```

5.8. If-then-else statement

The If then statement is used to perform actions if a certain condition is met. The condition is specified using OCL.

The else...endif portion of the statement is optional.

```
if (ocl invariant) then
  Action;
else
  Action;
endif;
```

5.8.1. Examples of if-then-else

Example 1

```
if (self.Amount > 200 and self.Creditor.Name->size() = 1) then
  self.Creditor.Name : "Major receiver";
else
  self.Creditor.Name : "Private receiver";
```

```
endif;
```

```
<Transaction>
    <Id>aaaaa</Id>
    <Amount>2</Amount>
    <Creditor>
        <Name>Private receiver</Name>
    </Creditor>
</Transaction>
```

5.9. Foreach action

The foreach action performs an action on a collection of XML elements. The collection is created using an OCL query.

Foreach is used to perform the same action on all child elements of the same name in a parent element (maxOccurs is greater than 1 for the child element).

```
foreach value in (OCL query)
  value.Action;
endeach;
```

5.9.1. Examples of foreach

Examples use the context *Message*

Example 1: deleting all transactions

```
foreach t in (self.Transaction)
  t.delete();
endeach;
```

```
<Message xmlns="http://www.XMLdation.com">
    <Header>
        <Id>ExampleId11</Id>
        <TimeStamp>2015-07-03T12:17:50</TimeStamp>
        <ControlSum>6</ControlSum>
        <NumberOfTransactions>3</NumberOfTransactions>
    </Header>
</Message>
```

Example 2: defining the range of foreach

```
foreach i in (Set{1 .. 10})
  auto tx : createNode("TransactionType1");
  tx.Id : i.toString();
  self.Transaction.add(tx);
endeach;
```

```
    <Transaction>
        <Id>1</Id>
    </Transaction>
    <Transaction>
        <Id>2</Id>
    </Transaction>

   (...)
```

Example 3: Modifying values in every Transaction

```
auto i : 0;
foreach t in (self.Transaction)
  t.Id : "Overwritten Id " + i.toString();
  t.Amount : 200;
  i : i + 1;
endeach;
```

```
    <Transaction>
        <Id>Overwritten Id 0</Id>
        <Amount>200</Amount>
        <Debtor>
            <Name>Debtor1</Name>
        </Debtor>
        <Creditor>
            <Name>Creditor1</Name>
        </Creditor>
    </Transaction>
```

5.10. Seed data

Seed data stores data in CSV-format and can be accessed and modified within myXML. Seed data is accessible via actions and therefore can be used to insert data to XML files. Seed datas are mostly used in

conjunction with foreach loops.

foreach item in (<name of seed data>.allInstances())

self.child : item.value;

endeach;

5.10.1. Examples of seed data

Seed data in these examples called "namelist" and contains following data:

firstname; lastname
Sherell; Christo
Hollis; Titus
Akilah; Perales

Example 1: Replacing existing data with values from seed data and generated content

```
// Deleting existing data
foreach t in (self.Transaction) t.delete(); endeach;

// foreach loop for every row in seed data
foreach n in (namelist.allInstances())

  // Creating types
  auto tx : createNode("TransactionType1");
  auto party : createNode("PartyIdentification");

  // Generating dummy-data
  tx.Id : String.shortUniqueTimeStamp();

  // Fetching data from seed data, adding it to party name
  party.Name : n.firstname + " " + n.lastname;

  // Adding created party to Creditor
  tx.Creditor : party;

 // Adding transaction to the file
 self.Transaction.add(tx);

endeach;
```

```
<Transaction>
   <Id>AF9CW</Id>
```

```
    <Creditor>
      <Name>Sherell Christo</Name>
    </Creditor>
  </Transaction>
  <Transaction>
    <Id>z4pvR</Id>
    <Creditor>
      <Name>Hollis Titus</Name>
    </Creditor>
  </Transaction>
  <Transaction>
    <Id>F9gQi</Id>
    <Creditor>
      <Name>Akilah Perales</Name>
    </Creditor>
  </Transaction>
```

Note that this data is not schema valid.

5.11. Save() - creating multiple files

It is possible to create multiple files by making an individual action for each file to be created. Alternatively, save() method combined with foreach() can be used in one action to create multiple files without the need to create individual action for each file to be created.

Method save() takes two parameters, type and name. Type indicates what type (and thus element) is being saved. The type used for the method should be the root type of the XML-file, therefore an action using save() should have the root as a context.

5.11.1 Example of save()

Example 1: Used within foreach()

```
foreach n in (Set{1 .. 10})
  auto newDoc : createNode(Message);
  newDoc.Header : createNode();
  newDoc.Header.Id : "value";
  save(newDoc, "name.xml")
endeach;
```

Following XML will be present in 10 files

```
<Message>
<Header>
<Id>value</Id>
```

```
</Header>
</Message>
```

5.12. Bad data

Creates XML that is not valid. There is three different ways to produce bad data.

1. Place bad data in primitives (force_bad_data)

2. Misspell XMLtags (force_misspell_xmltag)

3. Remove brackets (force_remove_xmltag_bracket)

5.12.1 Example of force_bad_data

Example 1: Force wrong value to datetime element

```
self.TimeStamp : force_bad_data("THIS IS A BAD DATE");
```

```
<Header>
    <Id>ExampleId11</Id>
    <TimeStamp>THIS IS A BAD DATE</TimeStamp>
    <ControlSum>6</ControlSum>
    <NumberOfTransactions>3</NumberOfTransactions>
</Header>
```

5.12.2 Example of force_misspell_xmltag

Example 1: Force wrong element name

```
force_misspell_xmltag("badtag1",self.Header);
```

```
<Message xmlns="http://www.XMLdation.com">
  <badtag1>
    <Id>ExampleId11</Id>
    <TimeStamp>2015-07-03T12:17:50</TimeStamp>
    <ControlSum>6</ControlSum>
    <NumberOfTransactions>3</NumberOfTransactions>
  </badtag1>
```

5.12.3 Example of force_remove_xmltag_bracket

Example 1: Remove bracket from Header element

```
force_remove_xmltag_bracket(self.Header);
```

```
<Message xmlns="http://www.XMLdation.com">
    Header>
        <Id>ExampleId11</Id>
        <TimeStamp>2015-07-03T12:17:50</TimeStamp>
        <ControlSum>6</ControlSum>
        <NumberOfTransactions>3</NumberOfTransactions>
    </Header>
```