**ANSIBLE DSL FILES ORGANIZATION**

The Ansible DSL + Editor consists of 5 different projects generated by xtext. All the files listed here can be found in the “src” subfolder of the associated project.

* org.sodalite.dsl.ansible:
  + AnsibleDsl.xtext: The grammar, consisting of the production rules.
  + AnsibleDslGenerator.xtend: The generator, which generates the playbooks yaml files from the abstract Ansible models.
  + AnsibleDslScopeProvider.xtend: The scope provider, which implements the code that makes the scoping work.
  + If looking in “src-gen” instead of “src”, the model (consisting of various Java classes) generated from the xtext grammar can be found.
* org.sodalite.dsl.ansible.ide: Nothing was implemented here.
* org.sodalite.dsl.ansible.tests:
  + AnsibleDslParsingTest.xtend: Contains the tests of the generator.
* org.sodalite.dsl.ansible.ui:
  + AnsibleDslProposalProvider.xtend: The proposal provider, which implements the code that creates the suggestions for the user in the editor.
  + AnsibleDslProjectTemplateProvider.xtend: Implements the Eclipse wizard for creating Ansible DSL project. Contains also an “Hello world” template.
* org.sodalite.dsl.ansible.ui.tests: Nothing was implemented here.

**ANSIBLE METAMODEL**

A documentation about the metamodel is useful for the implementation in the Knowledge Base and for explaining the work done with the Ansible DSL, since the metamodel and the DSL were designed in parallel.

ABOUT THE NOTATION AND THE APPROACH

The “1 to 1” relations are all represented like attributes in the entities, to improve readability. The question mark “?” written before one of those attributes means that the relation is “1 to 0..1”. For example, the Base entity could be in relation with 0 or 1 Connection entity. In all the other cases, the relation is represented with the classic arrow with the indication of the cardinality of the relation.

**2 Metamodels will be presented:**

* the first one is simpler, and it is intended to be the one to be used as a reference for the implementation in the Knowledge Base. This metamodel is coupled with the model generated by the xtext grammar implemented in the IDE, in the sense that there is *almost* a 1 to 1 relation between the entities of this metamodel and the entities belonging to the model generated by the xtext grammar. However, there are some entities present in the model generated by the xtext grammar that are not considered really useful in the KB metamodel, so they were not modelled here.
* The second metamodel is also *almost* in a 1 to 1 relation with the xtext generated model, but it is closer to it. The entities that were not considered useful in the KB metamodel are instead modelled here to give to the reader a better understating of the work done with xtext.

The metamodels are split among different figures, to improve readability.

Because of the strong relation between the metamodels and the grammar, and because the Ansible grammar defined in the IDE is used together with the RM grammar defined in the IDE as well, these metamodels contains references to elements belonging to the RM grammar. When this happens, those elements are referenced with a “RM\_” prefix.

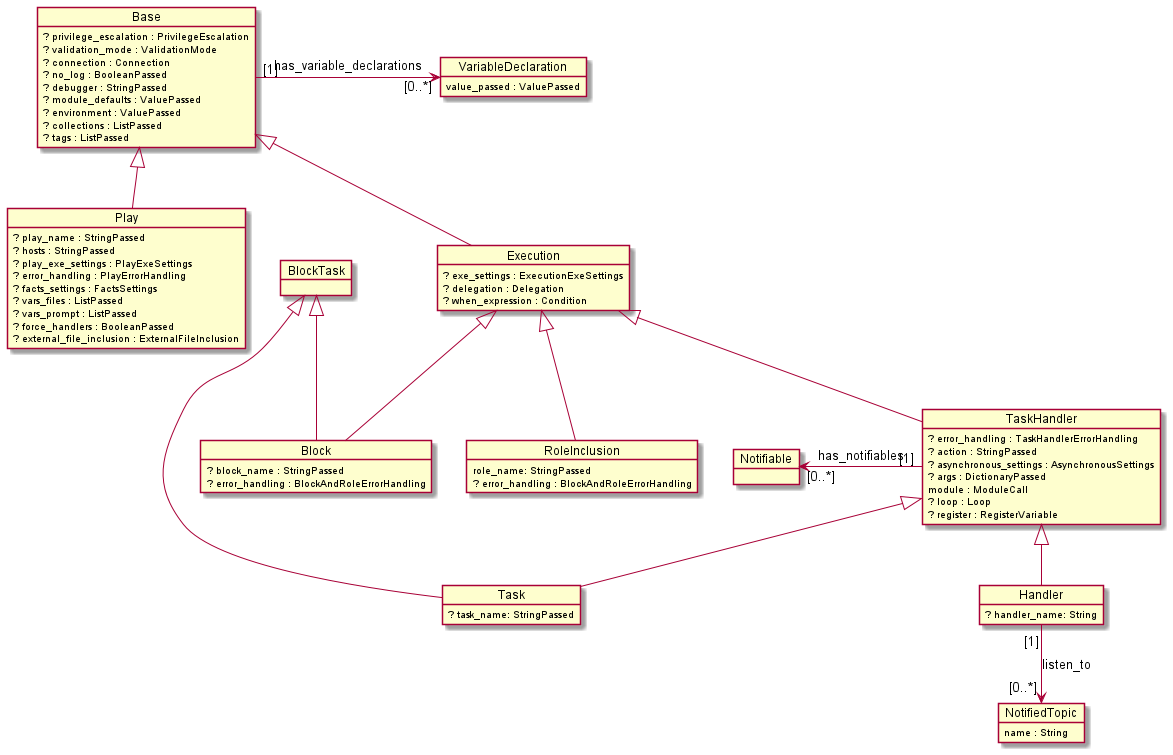
The main source used for understanding Ansible in order to build the metamodels was its [documentation](https://docs.ansible.com/ansible/latest/index.html).

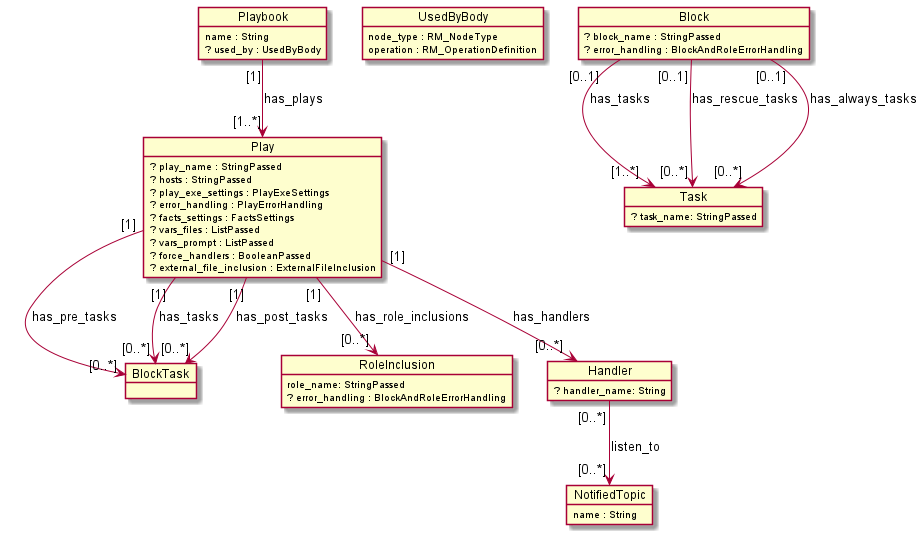
The 2 main points that these metamodels are based on are the following:

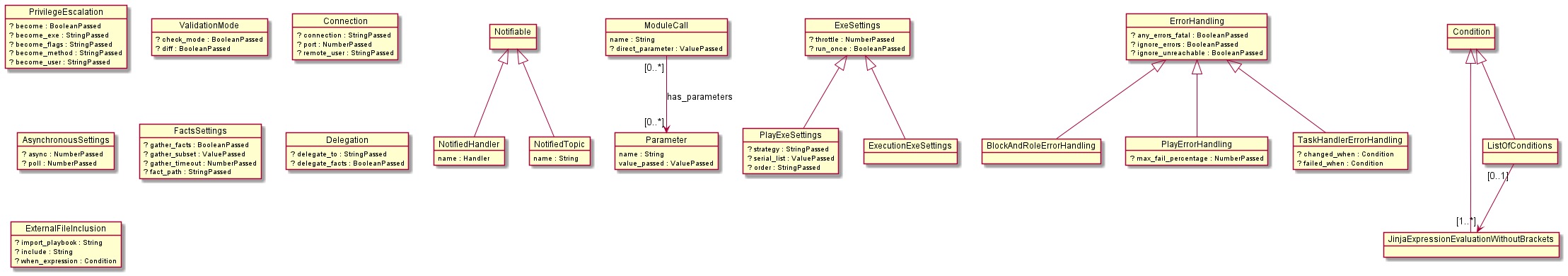
* **Group the attributes that are in common among different Ansible entities:** one of the main characteristics of Ansible is that its entities have many attributes that can be set by the programmer. The same attribute can be set in different entities, so in the metamodels we model entities that group those common attributes, and we make the “classic” Ansible entities inherit from them. As it will be better explained later, the Base entity is a good example of this: its purpose in this metamodel is just grouping in it all the attributes that can be defined in plays, blocks, etc., and then, as it can be seen in the first image, the entities that model plays, blocks, etc. just inherit from Base.
* **Introduce a new level of abstraction, not present in the real Ansible itself, to group attributes that are logically related to the same aspect**: when the programmer is defining the attributes of an entity in Ansible, they are “on the same level”, in the sense that there is not any distinction among them based on what they are defined for. It was found interesting, in these metamodels, to group them in terms of what they are used for. To do so, the main source of inspiration was of course the [documentation](https://docs.ansible.com/ansible/latest/index.html). It was not however the only source: the [Ansible repository](https://github.com/ansible/ansible) was useful as well, not only for discovering all the attributes belonging to the language, but also to understand how to logically divide them, since in the code of the repository there are some comments that help understanding it. For example, [this one](https://github.com/ansible/ansible/blob/devel/lib/ansible/playbook/base.py#L622) makes really explicit the fact that the attributes become, become\_method, become\_user etc. are used to set the privilege escalation. So, an entity PrivilegeEscalation was defined in these metamodels and those attributes are grouped in it.

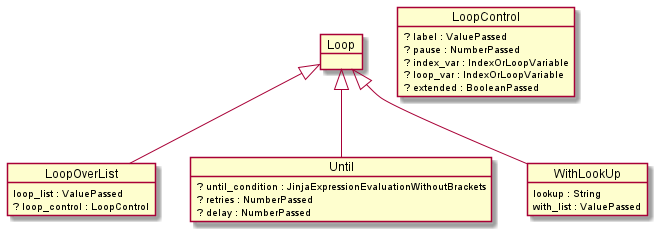
THE FIRST METAMODEL (THE ONE TO BE USED IN THE KB)

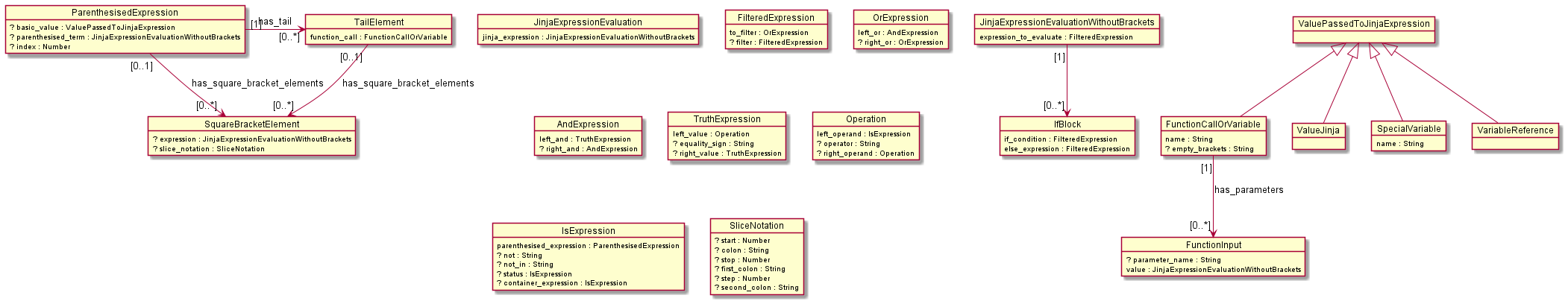
In the following the figures reporting the first metamodel are shown, with an explanation of the main aspects of it.

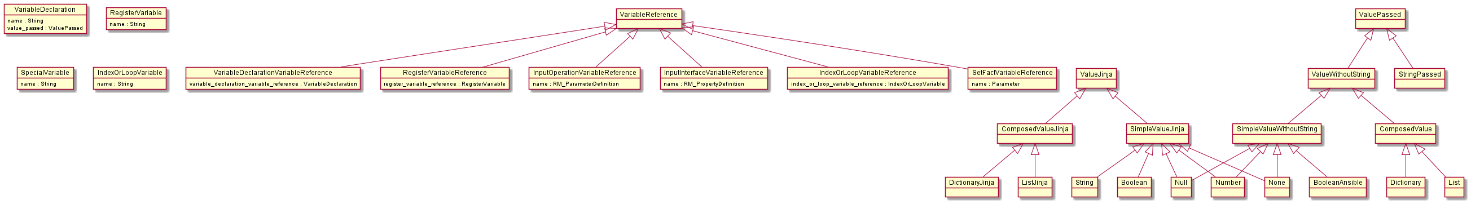


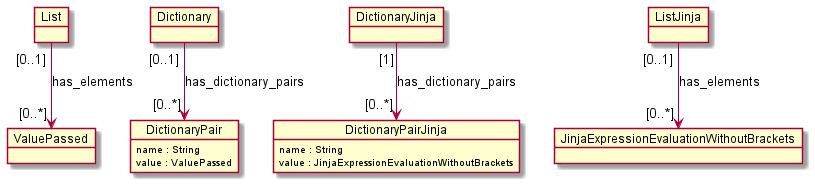


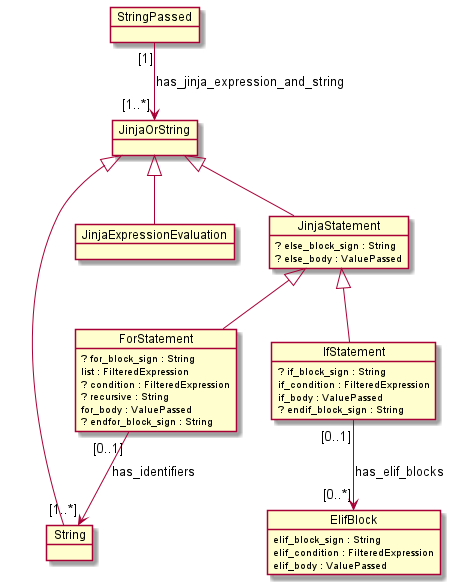


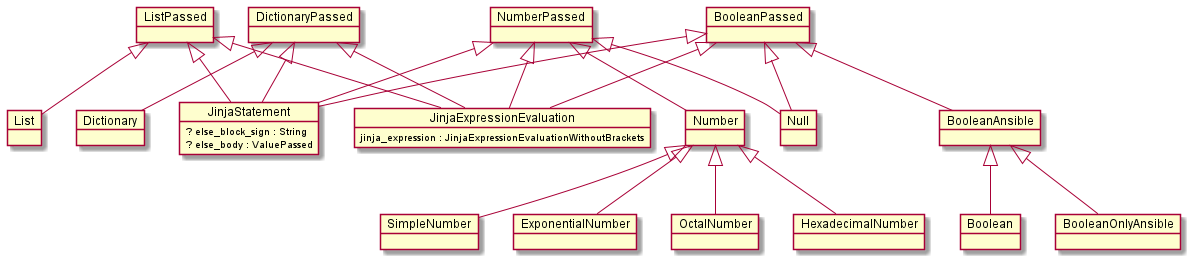












* Base: This entity was modelled by taking inspiration from the official ansible repository (<https://github.com/ansible/ansible/tree/devel/lib/ansible/playbook>) All the main entities of the model inherit from the Base entity, which groups the attributes that they have in common.
* Play: It all starts with the Playbook, which is a collection of plays (as it can be seen in the second image). The playbook is the artifact to which the interface’s operation of the TOSCA Resource Model refers. The playbook can contain one or more plays, and each play contains an ordered list of operations (BlockTask) to be executed. The operations (BlockTask) could be “normal” operations, “pre\_tasks” or “post\_tasks”. It’s just a matter of order of execution: “pre\_tasks” are executed before the roles included, the “normal” operations after the roles included, then the triggered handlers and then the “post\_tasks” (if some concepts named here are not clear, the reader can find an explanation in the points that follow).  
  Play inherits from Base, so it contains all the Base’s attributes plus other attributes shown in the first image.
* UsedByBody: As just underlined, the playbook is the artifact to which the interface’s operation of the TOSCA Resource Model refers. UsedByBody, used by the Playbook entity, represents the bridge between the TOSCA Resource Model and the Playbook model. Inside it, the user writes the reference to the node type using the playbook and the specific operation using the playbook. This procedure will allow the user to write references to the variables given in input by the TOSCA operation, Through the InputInterfaceVariableReference and InputOperationVariableReference entities (VariableReference will be better explained later).
* BlockTask: The operations contained in a Play can be blocks or tasks. The BlockTask entity was modelled to capture in a unique entity all the operations that can be run in a Play, so Block and Task inherit from BlockTask.
* Execution: The Execution entity was modelled to capture the entities that represent some operation that can be run. The main point is grouping in this Execution entity all the attributes that they have in common (apart from the Base’s attributes of course, from which Execution inherits).
* Block: As it can be seen in the second image, the Block is a collection of tasks (an ordered list). Apart from the “normal” tasks that it can contain, it can also have an ordered list of “always” tasks and “rescue” tasks. The “always” tasks are run in any case, even in case of errors. The “rescue” tasks are run to recover from error states.
* RoleInclusion: a play could contain an ordered list of roles identifiers to be executed. Executing a role means executing a set of operations, and a role can be for example downloaded from internet (usually from Ansible Galaxy). For example, a role could be named “mongodb\_server”, and executing that role means making the host on top of which it is run a mongodb server. So, the “mongodb\_server” role contains all the operations that, if executed, make the host a mongodb server. RoleInclusion however does not model the role itself, but its inclusion in the play. So it contains the identifier of the role (the name attribute) and other attributes inherited from Execution that can be used to set how the inclusion is done.
* TaskHandler: Since Task and Handler entities share a lot of attributes, the TaskHandler entity was modelled to group all of them.
* Task: It is the basic operation that can be run in a Play. The main thing to consider is that it executes a module (the same happens with the Handler, so the ModuleCall is contained in the TaskHandler entity). A module is a predefined operation that can be run and needs some inputs to be executed. The task passes those inputs to the module and executes it (to make an analogy: it is like if the module was an implemented function, and the task calls this function passing it some parameters). A list of the available modules can be found here: <https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html>

The inputs passed to a module are modelled as parameters (with the Parameter entity), that have an identifier and a value passed. However, it can happen that the input to the module is given without an explicit parameter with an identifier. It happens for example with the [“shell” module](https://docs.ansible.com/ansible/2.9/modules/shell_module.html#shell-module), in which the shell command is passed without an explicit named parameter. The direct\_parameter attribute modelled in this metamodel has the purpose of being used in situations like this, instead of the Parameter entity.

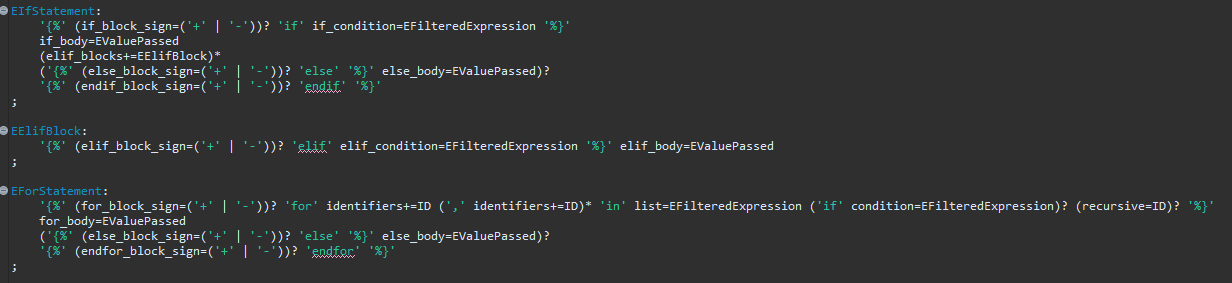
* Handler: Very similar to the Task, but it is executed when an event occurs. TaskHandler entity can be associated to some Notifiable elements, as it can be seen in the first image. A Notifiable element, as it can be seen in the third image, could be a Handler or a topic. So TaskHandler (tasks and handlers) can notify one or more handlers (or topics, as it will be explained shortly). It means that when the task/handler finishes its execution, the notified handlers are executed as consequence.  
  A Handler can also listen to a specific topic. If the Notifiable element associated to the task/handler is a topic, then all the handlers listening to that topic are executed consequently.
* Loop: tasks and handlers (so the TaskHandler entity) have the “loop” attribute. A Loop makes the execution of the task/handler iterate. As it can be seen in the fourth image, a Loop can be a LoopOverList entity, an Until entity or a WithLookUp entity. In the case of the LoopOverList entity, the task/handler is re-executed once for each element contained in the input given by the LoopOverList. The LoopOverList entity may also contain a LoopControl, with additional settings for the execution of the loop.  
  If the Loop is an Until entity instead, the task/handler is re-executed until a certain condition is met or a certain number of times.  
  The WithLookUp entity is meant to be used to handle situations in which, instead of the “classic” LoopOverList entity, the user decides to use the “with\_<lookup>” notation, which leads actually to similar effects of the LoopOverList one. The [specific section](https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#migrating-from-with-x-to-loop) of the Ansible documentation explains how various lookups could be replaced by the notation modelled with LoopOverList. However, WithLookUp was still modelled here to give to the user the possibility to use it. The “lookup” attribute is the string put after the “with\_”, so if for example the lookup attribute is “items”, the generated command will be “with\_items:” followed by the value passed to it, which is the attribute “with\_list” of the WithLookUp entity.
* JinjaExpressionEvaluation: in Ansible, to pass for example a variable as parameter to a module to be executed, its value needs to be accessed through a Jinja2 expression. This is done by delimitating the expression between double curly braces. The curly braces are not necessary in the cases in which the expression is evaluated to check if a certain condition is met (like for example in the “when” attribute), so in those cases the JinjaExpressionEvaluationWithoutBrackets entity is used.

The Jinja2 expression is modelled to be a FilteredExpression concatenated with a chain of 0 or more instantiations of the IfBlock entity. A FilteredExpression entity can contain filters applied to the expression before being evaluated, while the IfBlock models the classic “if…then…else” schema.  
A Jinja2 expression can contain various elements and operands, like “and”, “or”, “>”, “<”, “+”, “-“, etc. In the xtext grammar this is defined thanks to various correspondent productions with right recursions, and this finds correspondence in the metamodel. The basic element of the chain of right recursions is the ValuePassedToJinjaExpression entity, while at the top of the chain we find the JinjaExpressionEvaluationWithoutBrackets entity itself.

* Jinja2 expressions are not the only feature used by Ansible to pass values. Jinja2 statements are also used, and they are modelled with the JinjaStatement entity. Instead of the double curly braces, the Jinja2 statements are delimited by a curly brace and a “%” symbol (like “{% jinja statement %}”).  
  In this metamodel a JinjaStatement can be an IfStatement or a ForStatement. IfStatement offer the possibility to model an “if..then..else” schema with in addition 0 or more instantiations of the ElifBlock, which is basically the classic “elif” in the schema.  
  The ForStatement instead can be used for modelling for cycles. The can also have an if condition (the “condition” attribute), an “else” put at the end of the cycle (since both IfStatement and ForStatement can have an “else”, the “else\_body” attribute is put inside JinjaStatement, since the both inherit from it). The ForStatement can also contain the “recursive” keyword, which is a string and is modelled through the “recursive” attribute. ForStatament contains 1 or more identifiers, and this is modelled through the “has\_identifiers” relation. For example, if the cycle is like “for i, j in list do something”, then the identifiers are i and j.

Jinja2 statements can contain a “+” or a “-“ after the “%” symbol, and this is modelled with the various block\_sign attributes.  
the “bodys” (or, in other words, the “do something” that you would write in a for block or in an if…then…else block) have ValuePassed as type: the same used for the Parameter entity in the modules.

Since the UML model does not shows in a clear way the order in which the attributes should be defined by the user, the following image aims at giving some clarification about that. It is a screenshot taken from the xtext grammar, showing the associated production rules:



May the reader know that the model generated from the xtext grammar considers as strings all the attributes with the type “ID”, like for example “recursive”, so we did the same in the metamodel shown in the various figures.

* TailElement: an expression could have a tail: basically a “.” followed by a function or the identifier of an inner element, if the expression represents a dictionary. TailElement is the entity put after the “.” symbol. Of course, the expression could have a chain of tail elements.  
  It is also possible, for each tail element, to access a specific element of a list (if the overall expression represents a list), so there is also the SquareBracketElement entity. TailElement could be in relation with 0 or more instantiations of SquareBracketElement. For example, if we have this: “variable.inner\_element[2][3]”, then the tail “inner\_element” is in relation with the 2 square bracket elements “[2]” and “[3]” (in this example, “variable.inner\_element” is a list, and “variable.inner\_element[2]” is also a list).  
  ParenthesisedExpression has also the same relation with SquareBracketElement because the expression could be itself a list and we may want to access an element of it, so without any tail element involved (like “variable[4]”).  
  As it can be seen, the SquareBracketElement contains 2 attributes: “expression” and “slice\_notation”. The figure doesn’t make it clear, but they are mutually exclusive and one of them must be present. The “expression” attribute, which is of type JinjaExpressionEvaluationWithoutBrackets, captures the examples just described, in which we talked about a number between the square brackets. However, there are other possibilities: it could be string (a key of a dictionary), a variable reference (a variable could contain for example a number or a string, so it can be used here), a function returning a number or a string. All those elements can be defined thanks to the JinjaExpressionEvaluationWithoutBrackets entity.  
  If “slice\_notation” attribute is used instead, the classic [slice notation](https://stackoverflow.com/questions/509211/understanding-slice-notation) is being used. Since the model in the figure just shows that all the attributes in the SliceNotation entity are not mandatory, showing a screen from the xtext grammar is necessary:



The production rule shows that there are 2 possibilities: either there is one colon (the “colon” attribute) or there are 2 colons (the first and second colons). It also shows the criteria by which the other attributes can be used.

* ValuePassedToJinjaExpression: This is the basic element of a Jinja2 expression.
  + ValuePassedToJinjaExpression could be a value (ValueJinja: it will be better explained later). A value, as it can be intuitively be thought, is something like a number, a string, a dictionary, a list, etc.
  + ValuePassedToJinjaExpression could be a SpecialVariable. This term was chosen by taking inspiration from the [specific page](https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html) of the Ansible documentation. It captures the variables that contain something related to the internal state of Ansible. To make it clearer, an example could be “ansible\_facts”. Facts are basically properties gathered from the host on which the ansible playbook is run, like for example its IP address, and “ansible\_facts” is a variable (a dictionary) containing those facts. Even if not in the list of the special variables in the associated page of the Ansible documentation, it is reasonable to use the SpecialVariable entity also when the “item” variable needs to be used. “item” is a keyword in ansible for accessing an element of a list in a LoopOverList. It could be for example the case that we have a list of 3 elements, and we want to print each one of them. So ansible basically tells something like “print item” and this command is executed 3 times. Each time “item” will have the correspondent value of the element of the list.
  + It could be a function called, and this element is captured by the FunctionCallOrVariable entity. The function could take 0 or more inputs when called, defined with the FunctionInput entity. The input could be defined in 2 ways: with or without its explicit name. For example: “functionCalled(input1=”hello”, “world”)” has 2 inputs: the first one with the explicit name of the input (“input1”) and value “hello”, while the second with only the value “world”.  
    If the function doesn’t have any input, then it can have empty brackets or not. For example, “functionCalled” and “functionCalled()” are 2 possibilities. If the empty brackets are there, “empty\_brackets” attribute is defined, and it actually consists of the string “()”.  
    FunctionCallOrVariable entity is called like this for a specific reason. As just explained, it can be used for defining a function called, but this is not the only possibility. VariableReference entity is the one to be used when referring to a defined variable, but it may be possible for any reason that the user wants to reference a variable that is not defined anywhere. For example, it may be that the user wants to reference a variable given as input by an operation of an interface of a TOSCA Resource Model. To do this, the Playbook model and the Resource Model need to be connected through the UsedByBody entity. This connection may be not set up for any reason. For example, the Resource Model was not defined yet, but the user already has ideas about how to define it and knows which inputs it will pass to the Playbook. If the user wants to define the Playbook and just write the name of the variables that he/she knows will be in the Resource Model, this is possible. The attribute “name” of FunctionCallOrVariable can be the name of the function or the name of a variable. This mechanism was adopted, instead of defining another entity for the purpose, to avoid ambiguities in the grammar and to keep it simple.
  + Finally, ValuePassedToJinjaExpression could be a VariableReference entity, which is a reference to a variable. The following dedicated point is necessary to explain better this entity.
* VariableReference: it represents a reference to a variable. There are different types of variables, in the sense that there are different possible ways in which they are declared/passed to the playbook. Thus, there are different types of VariableReference entities:
* VariableDeclarationVariableReference: it could be identified as a reference to the “classic” declared variable: the VariableDeclaration entity associates an identifier of the variable to a value, and VariableDeclarationVariableReference is a reference to VariableDeclaration.
* RegisterVariableReference: it is a reference to RegisterVariable, an entity that models the variables that are produced in ansible after the execution of a task or a handler: after the execution, the output of it can be stored in a new variable, the registered variable.
* InputOperationVariableReference and InputInterfaceVariableReference: They are both related to variables given in input from the TOSCA Resource Model to the playbook. The main difference between them is that the variable given in input could be defined in the “operation” section of the interface, like it’s done [here](https://github.com/SODALITE-EU/ide/blob/master/dsl/org.sodalite.dsl.examples/snow_split.v2/snow_v2.rm#L19) for “remote\_server” and mysql\_db\_pass” (in that case we have the InputOperationVariableReference) or in the interface itself, like it’s done [here](https://github.com/SODALITE-EU/ide/blob/master/dsl/org.sodalite.dsl.examples/hpc/hpc.rm#L87) for “wm\_public\_address”, “wm\_username”, etc. (so we have the InputInterfaceVariableReference). The main reason for having these 2 different entities, instead of just one capturing both the cases, is that in the first case the RM grammar uses the ParameterDefinition entity to define the input, while in the second case it uses the PropertyDefinition entity. Since the inputs are defined in these 2 different ways, it was reasonable to define 2 different correspondent entities for referencing them.
* IndexOrLoopVariableReference: while looping, it’s possible to define new variables with the “[index\_var”](https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#tracking-progress-through-a-loop-with-index-var) keyword and the [“loop\_var”](https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#defining-inner-and-outer-variable-names-with-loop-var) keyword. This entity is just used for capturing both the cases.
* SetFactVariableReference: there is a particular module, named “set\_fact” that allows to actually declare a new variable. This entity allows to refer to variables defined in this specific way. The attribute “name” is a Parameter because the parameters passed to the “set\_fact” module represents the new variables declared.
* ValuePassed: First, it is necessary to explain what a value in this model is. It can be a simple value (string, boolean, number, null) or a composed value (a list, a dictionary).  
  JinjaExpressionEvaluation and JinjaStatement, explained before, were introduced to model the values passed to a module to be executed, but it is not enough alone. As it can be seen in [this example](https://github.com/SODALITE-EU/iac-management/blob/master/use-cases/snow-uc/snow-openstack/playbooks/docker/create_docker_host.yml#L67), a value passed could be a JinjaExpressionEvaluation concatenated with a string. It could actually be that JinjaStatement is concatenated as well. So, it was introduced the JinjaOrString entity, that can be a JinjaExpressionEvaluation, a JinjaStatement or a string. StringPassed, instead, is an ordered set of JinjaOrString entities. In this way, we have the StringPassed entity that models the concatenation of strings, jinja expressions and jinja statements.  
  To sum up, ValuePassed could be a StringPassed or a value. Because StringPassed contains strings, this would lead to ambiguity in the grammar associated to this model, since both a hypothetical “Value” entity and StringPassed could be a string. For this reason, ValueWithoutString was introduced in the model, which represents any kind of values (boolean, dictionary, etc.) but it cannot be a string.

For what concerns the attributes in general, like for example “delegate\_facts”, “async”, “check\_mode”, etc., the value that is passed to them is not just simply ValuePassed. The point here is that, thanks to the [documentation](https://docs.ansible.com/ansible/latest/index.html) and the [Ansible repository](https://github.com/ansible/ansible), we know in advance what is the type expected from each attribute. If, like in the case of [“check\_mode”](https://github.com/ansible/ansible/blob/devel/lib/ansible/playbook/base.py#L613), the value expected is a boolean, what can actually be passed to “check\_mode” is a boolean, a jinja expression or a jinja statement (which should at run time be a boolean, but because of the strong relation between this metamodel and the xtext grammar defined in the IDE, here we are focusing rather on the compile time). So BooleanPassed is the entity that can be a Boolean, a JinjaExpressionEvaluation or a JinjaStatement, and is the one passed to “check\_mode”. In this way, if the programmer writes something like “check\_mode: 5”, the IDE will raise an error message, saying that the number type is not compatible with “check\_mode”. The same consideration holds for DictionaryPassed, ListPassed, NumberPassed. For what concerns String, StringPassed is already fine for this purpose, so that can be used instead of defining another StringPassed entity.

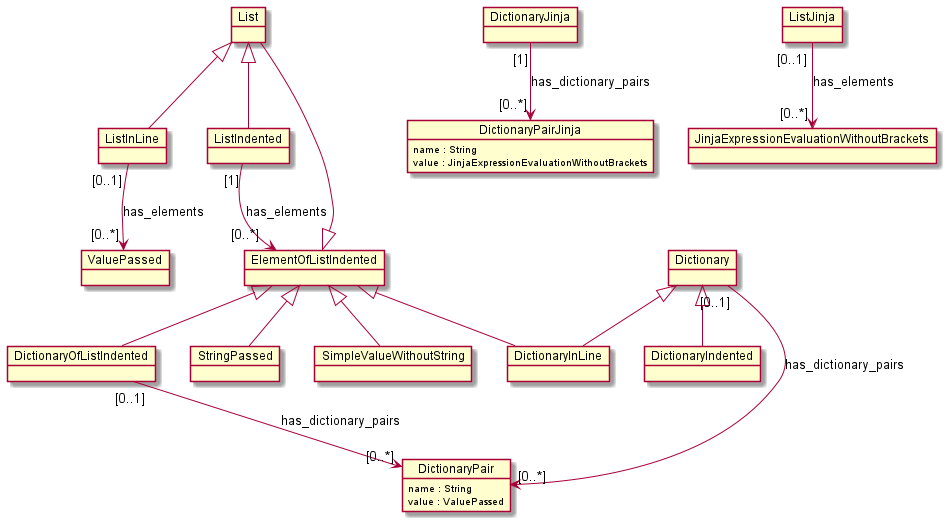
This kind of approach, that aims at forcing the programmer to pass the values with the right type, was not used for the Parameter entity (recall: it is the one involved in the ModuleCall entity) simply because in general we do not know what is the right type of value that should be passed as a parameter to a general module. We could know it if we consider a specific module: if we select one from the [list of modules](https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html) of the Ansible documentation, we can see the list of its parameters and for each one the expected type. If the IDE can access in some way (with some endpoints for example) a component that provides this information, then in principle it is possible to use an approach similar to the one just described for “check\_mode” for the parameters of the modules. However, this metamodel is general and does not take into account this possibility. In general, one could, with the ModuleCall entity, “call” every possible module (even not existing in the [list of modules](https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html)) by providing an identifier, and could pass every parameter that he/she wants, by providing an identifier. So, of course, under these circumstances it is inevitable to let the programmer use every possible type, with the ValuePassed entity.

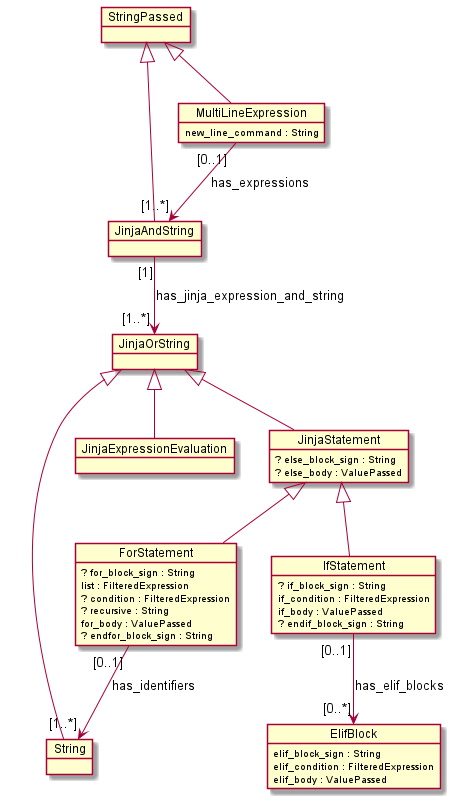
* ValueJinja vs. ValuePassed: It is important to underline the following: ValuePassed could contain jinja expressions, but a jinja expression is not supposed to be used inside another jinja expression. So, to consider the values that are passed to a jinja expression it was necessary to model another new entity: ValueJinja. It was already specified that ValuePassedToJinjaExpression could be a ValueJinja, now the reason why it is ValueJinja instead of ValuePassed should be clear.

One of the main differences between ValueJinja and ValuePassed is how the strings are handled. It was already explained that in ValuePassed the are contained in StringPassed, so ValueWithoutString can not be a string itself. In ValueJinja this problem does not exist: StringPassed is not involved, as it could be a jinja expression or a jinja statement, so String is simply one of the possible choices for SimpleValueJinja (which inherits from ValueJinja, as shown in the figure).  
Another key difference is represented by the dictionaries and the lists. With ValuePassed, A list contains, as elements, other ValuePassed elements. Dictionaries can as well contain, in DictionaryPair, ValuePassed as value. With ValueJinja instead, this would not be possible: a list or a dictionary containing ValuePassed instances could contain jinja expressions or jinja statements. ListJinja and DictionaryJinja were model to solve this problem: instead of ValuePassed instances, they contain JinjaExpressionEvaluationWithoutBracket instances.

THE SECOND METAMODEL

This metamodel is more similar to the one generated by the xtext grammar. However, the differences with respect to the first one are just a few, so here only those differences are reported.





* Dictionaries and lists: in the xtext grammar it is possible to define lists and dictionaries in line or indented. For example:  
  **{key1: “hello”, key2: “world”}**  
  is a dictionary in line.

The following instead:  
 **key1: “hello”  
 key2: “world”**

is a dictionary indented.

In the same way, the following:  
**[1, 2, 3]**

is a list in line, while:

* **1**
* **2**
* **3**

is a list indented.

There is not a reason to distinguish the 2 notations in the Knowledge Base, that is why in the first metamodel it was chosen not to model this distinction, while it was necessary in the xtext grammar.

ElementOfListIndented was introduced simply because DictionaryIndented was giving problems in xtext because of the combination of the indentation of the list and the indentation of a dictionary being an element of the list. So, specifically in the case in which ListIndented is used, ElementOfListIndented is the entity representing an element of it. It is more or less like ValuePassed, a part from the thing about the dictionaries just explained.

* MultiLineExpression: Ansible playbooks are usually written as yaml files, and yaml allows to write strings over multiple lines, like it is done [here](https://github.com/SODALITE-EU/iac-modules/blob/master/hpc/data/mover/playbooks/create-text-file.yml#L5). MultiLineExpression is the entity with the purpose of allowing the user to do so. The attribute “new\_line\_command” is used to define the command indicating that is a MultiLineExpression instead of a normal string: It could be “>” or “|”.  
  Another new entity, JinjaAndString, is the one containing a list of instances of JinjaOrString, instead of StringPassed. In this metamodel, both JinjaAndString and MultiLineExpression inherit from StringPassed.  
  Every line of the MultiLineExpression entity is an instance of JinjaAndString, so it is a concation of strings, jinja expressions, jinja statements.  
  It was thought that it was not necessary to introduce MultiLineExpression in the metamodel of the Knowledge Base, since it is basically like StringPassed of the first metamodel, but with multiple lines. Also, in yaml this case is considered as a normal string.