Value Type Modelling

CAMEL includes the coverage of the type aspect. This coverage is twofold: (a) on the one hand, single values, either numeric or non-numeric, are captured; (b) on the other hand, the value types of such values can be also modelled. The value types can be used to denote the range of values that metrics (variables) can take. This is important for two main reasons: (i) to inspect whether the measurements of metrics are correct / valid, mapping to their value type, especially when they are produced by error-prone sources of information, like humans or sensors; (ii) to properly formulate the Constraint Problem (CP) model for deployment reasoning in the context of metric variables and the constraints that can be posed on them (where metric variables should be mapped to CP variables that should have a proper domain of values). Thus, value types, in the end, constitute important structured elements in CAMEL which should be associated with other critical elements, related to the measurement of applications or the evaluation of their utility functions. In the following, we exemplify what are the main value types that can be specified as well as their respective content which maps to the specification of values (either bound or member ones).

We follow a bottom-up analysis and we start first by explaining how the parts of types, i.e., their values can be specified in CAMEL. CAMEL enables the specification of non-numeric values, booleans and strings in particular, as well as numeric ones, which take the form of int, float and double values. In contrast to how other CAMEL elements are specified, values can be described without mentioning any identifier for them. Their form has been actually confined to the minimum in order to limit the modelling effort of the user/modeller. We now attempt to explain how CAMEL is used to define such values through the supply of respective examples. We try to cover all kinds of values to be as exhaustive as possible.

Boolean values can be expressed through the supply of the respective boolean value, either true or false. The following example in CAMEL textual form showcases this:

```
boolean 'true'
```

In this example, we supply the boolean value of true. The value of false is denoted through the non-supply of the 'true' value. In other words:

```
boolean
```

String values can be expressed through the supply of the respective string. The following example denotes this:

```
string 'xyz'
```

Numeric values are expressed through supplying their (primitive) type and respective value. The following examples denote how all kinds of numeric values can be expressed.

```
int 1

float 2.0

double 3.0
```

There are 3 main value types as well as 2 auxilliary ones. The 2 auxilliary ones are the BooleanValueType and the StringValueType. Such types express the value type of booleans and strings. Such value types could be also used to express the domain of values of metrics / metric variables, although this might not so frequent, as metrics / metric measurements are usually numeric. Please note that the string value type encompasses any string value. In this sense, such a value type can be infinitely large to include any kind of string (value). On the other hand, if we desire to supply a restrained string value type that encompasses only a finite set of strings, then we should specify instead a List of strings, which maps to one of the 3 main value types in CAMEL.

A list is a collection of values of the same type. The type of such values can be simple or composite. In the first case, this means that the type can be basic/primitive, mapping to all possible strings, booleans, integers, floats and doubles. If the type is composite, this means that it should map to a ValueType in CAMEL (one of the 2 main ones, range and range union). Let us now supply some examples in order to exemplify these two cases in list modelling.

The first example enables the specification of a finite collection of strings. Such collection could take the form of: {"One", "Two","Three"}. Its respective specification in CAMEL would be the following

```
list myList{

        primitive type string  here we denote the primitive type of the list values

        values [string 'One', string 'Two', string 'Three']  here we define the values of the list

}
```

As it can be seen, we supply the primitive type to which the list values conform and then, in the next line, each value, confined between the enclosing brackets, based on the respective syntax of the corresponding kind of value.

The second example denotes the specification of a list of values that conform to the int range of [1,10]. We supply in the following fragment only the specification of the list. The int range value type will be supplied when CAMEL's range value type is explained in the sequel:

```
list myList2{
        type IntRangeOneTenInclusive  a reference to the value type of the list's values
        values [int 1, int 5, int 10]  the specification of the values of the list
}
```

Similarly to the case of the first list example, we refer first to the (now composite) type of the list values and then we define each of them withing enclosing brackets.

A *Range* in CAMEL represents a range of numeric values. As such, the range can map to either int, float or double values. This range is actually specified by explicating what is the primitive type of the range's values as well as the two main bounds of the range, i.e., the lower and the upper. Both bounds are specified through the supply of the actual bound value and a boolean value indicating whether the value is included or not in the range. The absence of a bound signifies infinity. In particular, the absence of the lower bound denotes negative infinity, while the absence of the upper bound denotes positive infinity. In the following, we supply some examples of ranges specified in CAMEL:

First range example: specification of the int range [1,10] as also utilised in a previous example (for the list of values). The following code fragment can be used to specify this range:

```
range IntRangeOneTenInclusive{
        primitive type int  reference to the primitive type of the range
        lower limit incl. int 1  specification of its included lower bound
        upper limit incl. int 10  specification of its included upper bound
}
```

Second range example: specification of the double range (0.0,+infinity) in CAMEL:

```
range DoubleZeroPositiveInfinity{
        primitive type double  reference to the primitive type of the range
        lower limit double 0.0  specification of the non-included lower bound of the range
}
```

As it can be seen, there are 3 main differences with respect to the first example: (a) the primitive type is now double; (b) the lower limit is not included and maps to a double value; (c) an upper bound is not specified to denote positive infinity.

Finally, a range union, as its name witnesses, represents a union of ranges. Such ranges should not be overlapping. This means that they should not have any value in common. In addition, all ranges should have the same primitive type. Such a value type could map to the specification of metrics or metric variables which do not have a continuous domain of values. To exemplify the specification of a range union, we rely on the following example: a range union of [1,10] U [20,+infinity). The following textual CAMEL fragment denotes its specification (along with the specification of the second range, the first has been already supplied above):

```
range IntRange20PositiveInfinity{
        primitive type int  reference to the primitive type of the range
        lower limit incl. 20 specification of the included lower bound
}
range union OneToTenToPositiveInfinity{
        ranges [IntRangeOneTenInclusive,IntRange20PositiveInfinity]  reference to all the ranges included in the union
}
```

As it can be seen, the range union is just defined by referring to all the ranges that it includes. There is actually no need to specify the primitive type of this range union as this can be derived from its included ranges. An OCL rule is also involved to enable checking and validating that a unique primitive type characterises all the ranges of the union.