

Design Space Exploration Tool

“DESERT”

**Endre Magyari, ISIS
2002 April**

Interface models

The DESERT defines two APIs, one for submitting the data into DESERT, the *Input Interface*, and another one for dumping the output, the pruned Design Space, the *Output or Back Interface*. These interfaces and APIs are provided by the UDM Framework. The DESERT expects an input data network based on the input interface model, and outputs an output data network based on the output interface model. The input and the output data networks may persist on any of the supported UDM backends. Both the input and the output interface are defined in form of UML class diagrams.

Input Interface model

There are three different basic terms: **Space**, **Domain**, and **Property**.

Spaces

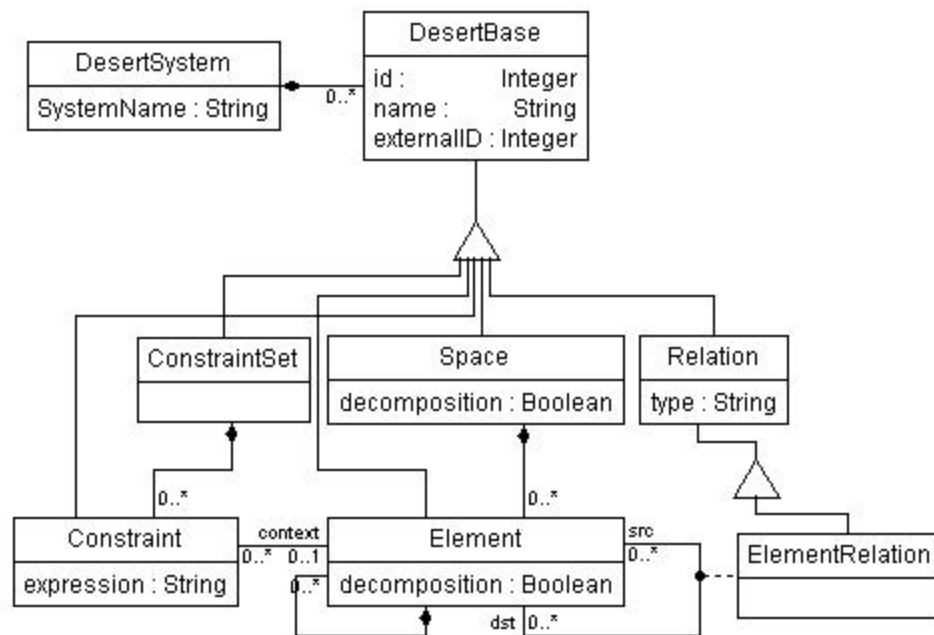


Figure 1 Input Interface Model - Hierarchical space

By *space* we mean an **AND-OR-LEAF** tree of elements. This, from the constraint satisfaction tool's point of view means that the **OR** nodes are variables, and their possible values are their children. The type of the node it's controlled by it's *decomposition* (Boolean) attribute and the number of sub-elements(children) of a node in the following way: }

- If the number of children is = **0**, then the type is **LEAF**.
- If the number of children is > **0** and *decomposition*, then the type is **AND**
- If the number of children is > **0** and *not decomposition*, then the type is **OR**

All the nodes may have *Constraints*. Each *constraint* has a *context* associated, which is a node of any type, and the *Constraints* are contained in *Constraintset*.

Constraintsets are there just for the reason to provide a mechanism for grouping the *Constraints* in several categories, which are specific to application. Well, partially true: There is one special *Constraintset*, the one with *name* “Implicit Constraints”. The *Constraints* in this constraint set are automatically applied and not shown on the DESERT user interface. The *expression* attribute is the constraint expression, in an extended OCL language. This will be discussed later in this document.

ElementRelations are permitted between the element to pass some extra information to the tool. These relations might be used when composing *Properties*.

DesertBase is an abstract base class; each class in the input interface model is directly or indirectly inherited from this class. The value of the *id* attribute should be unique for each UDM object passed to DESERT. The *name* attribute is used in the constraint expression; so, they should be unique as well at a certain level in the hierarchy. The *externalID* attribute may carry extra information throughout DESERT about the object, which is application specific. The values if these attributes are preserved through DESERT, so this is how the application can identify and match the objects when reading them back from DESERT.

DesertSystem is just a container for everything in the input model. This is needed because of the nature of the UDM, which require everything to be contained in a “*RootObject*”.

Properties

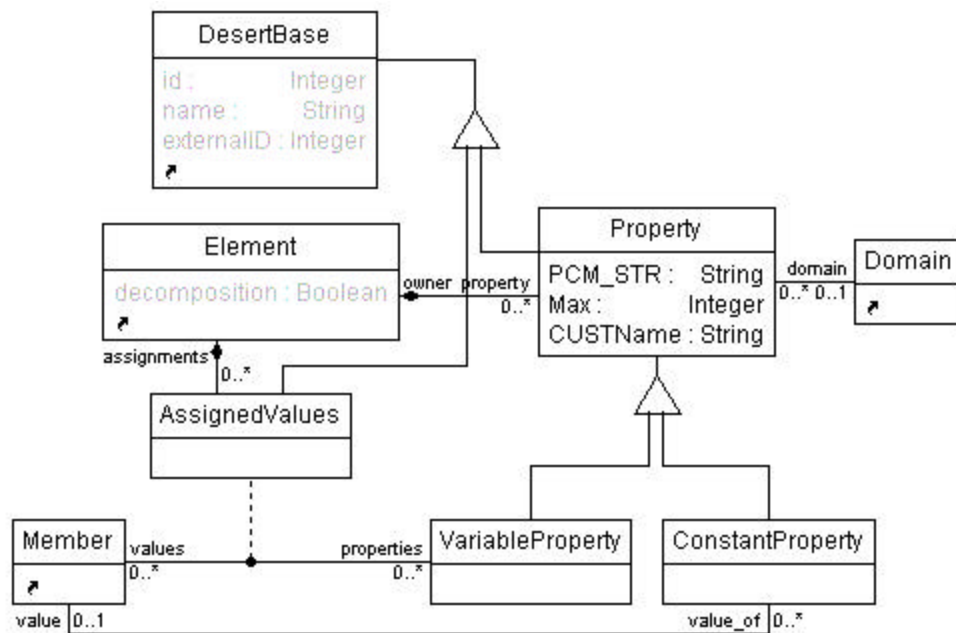


Figure 2 Input Interface Model - Properties

As mentioned above, leaf nodes can have *Properties*. These *Properties* then can be used to express *constraints*. A *property* is owned by an *element* and always belongs to a *domain*. This is the set of members that can be assigned to properties as values. The property values can be composed in the **AND-OR-LEAF** tree, if the *Property* to be composed is defined for all the leaf nodes in the sub-tree rooted at the node at which the property should be evaluated.

The **PCM_STR** attribute specifies how a *Property* should be composed, when the constraint engine needs to evaluate it at an **AND** or an **OR** compound node. The possible values for this attribute are:

- PCM_ADD** - specifies that the *property* is *additive*.
- PCM_MUL** - specifies that the *property* is *multiplicative*.
- PCM_AMED** - specifies that the *property* value of the compound node is equal to the *arithmetic median* of the same *property* values of the children.
- PCM_GMED** - specifies that the *property* value of the compound node is equal to the *geometric median* of the same *property* values of the children.
- PCM_MIN** - specifies that the *property* value of the compound node is equal to the *minimum* of the same *property* values of the children.
- PCM_MAX** - specifies that the *property* value of the compound node is equal to the *maximum* of the same *property* values of the children.
- PCM_NONE** - specifies that the *property* is it not a *composable property*. Thus, forcing the constraint engine to evaluate it at a compound node by adding such a constraint at a compound node will generate an error.

PCM_CUST - specifies that there will be a *custom function* linked against DESERT, which will compose this property whenever is needed. The *CUSTName* attribute will hold the name of the *custom function*.

There are two kinds of *properties*: *Variable property* and *Constant property*.

As one would easily guess, *Variable Properties* will be the variables of the constraint engine, thus, they may be assigned to more than one *member*. The assignments are done by the association class *Assigned values*, which are contained in elements. Each *Assigned value* object points to a *member* with it's *values* pointer and to a *Variable property* with it's *properties* pointer.

Constant properties are there to define *Properties* that are constant for a leaf node. Even if this is not used directly by the constraint engine, it's still needed when one needs to evaluate a property at a compound node, which implies that a certain property must be defined for each leaf node in the sub tree.

Again, everything is derived from the *DesertBase* abstract base class. One should pay extra attention to the *id* and *externalID* attributes of the *Assigned values*, because DESERT will return with set of *Configurations* which will contain a set of valid *Assignments*. In most of the cases the application will need to match the *Assignment* objects in the output data network with the *Assigned Values* objects in the input data network. This will be possible, since the *Assignment* object will preserve the *ids* of the *Assigned values* objects.

The abstract base class *Property* captures the common functionalities in *Variable properties* and *Constant Properties*, that these are contained in the *owner Element*, they operate on a *Domain*.

Domains

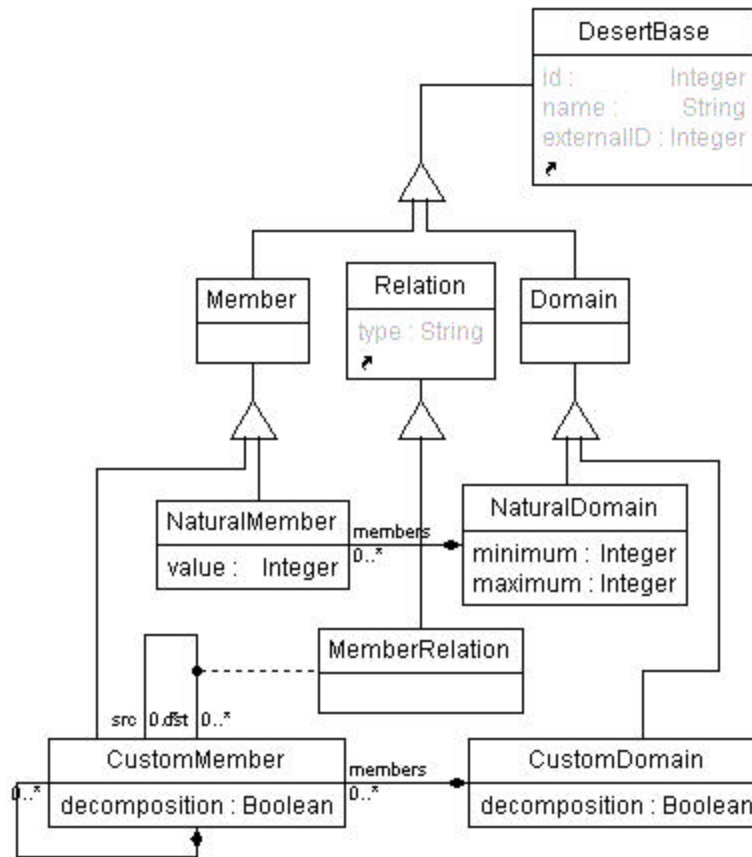


Figure 3 Input Interface Model - Domains

A *domain* is basically a set of *values*, which *properties* may be assigned to. There are two kinds of domains: *Custom domains* and *Natural Domains*.

The *Custom Domain* is again an *AND-OR-LEAF* tree and it has the same syntax as *Space*. However, the *decomposition* attribute is currently not used in the DESERT. The nodes of this tree are the *Custom members*, which act exactly the same way as *elements* in the *space*.

Member relations are there for the same reason element relations are there. One could use this information in a custom function when composing properties.

The *Natural Domain* is a set of natural numbers, which is defined by its attributes *minimum* and *maximum*. However, if one wants to actually assign a value from a natural domain, then one would need to create a *natural member* with that value in the *natural domain*.

The abstract base class *member* captures the common functionality of *Custom members* and *Natural members*. *Members* can be assigned as values to *Properties*.

The abstract base class *Domain* captures the common functionality of *Custom domains* and *Natural domains* that there are *Properties* assigned to them.

Output Interface model

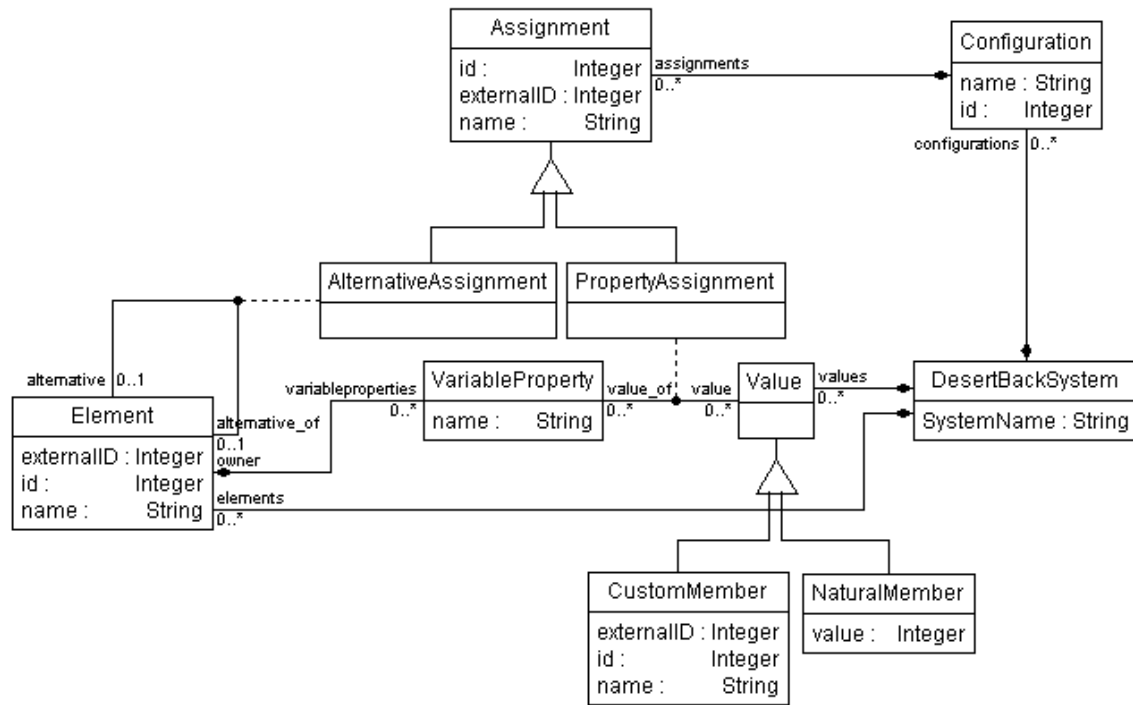


Figure 4 Output Interface Model

The above class diagram was designed based on the idea that DESERT outputs valid *configurations* that satisfy the applied *constraints*. Each *configuration* will have exactly one *Natural member* or *Custom member* assigned to each *Variable property*, and exactly one node (=Element) for each compound node of type **OR** in *space*.

The *DesertBackSystem* is a class that contains everything. This will be the type of the root object of the output data network. *DesertBackSystem* contains *values*, *elements*, and *configurations* that are objects of type *value*, *element*, and *configuration*, respectively. Further, the *elements* contain *Variable properties*. *Elements* and *Variable properties* have exactly the same semantics as in the *input interface model*.

A *Configuration* object identifies a particular configuration in the design space. The valid *configurations* are numbered, beginning from 1, and these are the values of their *id* attribute. The *name* attribute value will be set to “Conf. no. *x*”, where *x* is the *id* of the *configuration*. A *configuration* contains *assignments*, which can be of two kinds: *Property Assignment* and *Alternative Assignment*. *Alternative assignments* assign a compound **OR** node (*alternative_of*) to a child node (*alternative*). *Property assignments* assign a *Value* to a *Variable Property*. However, if there are multiple *Assigned value* objects in the input data network assigning the same *member* to the same *Variable property*, for each such *Assigned value* object there will be a *Property Assignment* object in the output data network.

The *Value* objects, and its derivatives, *Custom member* and *Natural member* correspond to the *member*, *custom member* and *natural member* objects, respectively, defined in the input interface model.

The values of the *id*, *externalID* attributes of the objects in the output data network are equal to the values of the *id* and *externalID* attributes of the corresponding objects defined in the *input interface model*.

The Constraint Language

The constraint language in DESERT is an extended OCL.

Command Reference for the extensions:

Global functions:

project() - returns the DESERT project

Project functions:

name() - returns the *Space* or *Custom domain* named “*name*”

Space and Custom domain functions:

children() - returns the collection of children(not recursive) nodes of a *Space* or a *Custom domain*

children(“*name*”) - returns the node named “*name*” from the children of a *Space* or a *Custom domain*

Custom member and Element (node) functions:

children() - returns the collection of children(not recursive) nodes of a node

children(“*name*”) - returns the node named “*name*” from the children of a node

Element functions:

implementedBy() - for an **OR** type node returns the selected *alternative*(:=the value) from its children.

name() - returns the value of the *property* named “*name*” for the node. If this *property* operates on a *Custom domain*, it will return a *Custom member*, if it operates on a *Natural domain*, it will return with an arithmetic value. If the context is a leaf node, it will directly read its *property* value, if it's not, it will return a composed value.

How to use it?

Desert is available in a binary distribution.

It contains the following files:

DesertTool.exe	- application executable
Desert.dll	- application extension
DesertD.dll	- application extension(DEBUG)
DesertIface.xml	- input interface model class diagram
DesertIfaceBack.xml	- output interface model class diagram
Xerces-c_1_2D.dll	- 3 rd party dll
Xerces-c_1_2.dll	- 3 rd party dll debug
Desert.pdf	- this documentation

DesertTool.exe is a standalone MFC application.

When invoking without command line parameters, it starts with a file open dialog for the input data network. Once the data read, a GUI comes up, where the constraints can be applied one by one. By exiting this application, another file open dialog comes up for the output data network. The configurations are saved.

DesertTool.exe also can be invoked with a single command line parameter, which would specify the input data network skipping thus the file open dialog for the input. After exiting, the output is also generated automatically and saved using the same UDM backend technology as the input data network, in a file with the same name as the input data network, but a “_back” is suffixed.

DesertTool also registers itself in the system registry, so you can invoke and make it auto-magically generate the output.

The code for this:

```
HKEY hKey;
DWORD dwSize = _MAX_PATH;
DWORD dwDataType = REG_SZ;
DWORD dwValue = 0;
char desert_path[_MAX_PATH];

if(::RegOpenKeyEx(HKEY_CURRENT_USER, "Software\\ISIS\\DesertTool\\Data", 0, KEY_QUERY_VALUE,&hKey) ==
ERROR_SUCCESS)
{
    if(::RegQueryValueEx(hKey,"Path", NULL, &dwDataType, (unsigned char *)desert_path,&dwSize) != ERROR_SUCCESS)
    {
        // Close key
        ::RegCloseKey(hKey);
        //handle registry error
        return;
    }
    // Close key
    ::RegCloseKey(hKey);
} //eo if RegOpenKeyEx is ok
else
{
    //handle registry error
    return;
}

//invoke DesertTool, and wait until it finishes
_spawnl(_P_WAIT, desert_path, desert_path, (LPCTSTR)str, NULL);
```

How to compile it?

Case Study: Desert in MILAN