A Generative Approach to the Presentation and Manipulation of Hierarchical Data

Joni Helin

VTT Information Technology
Sinitaival 6, P.O.Box 1206, FIN-33101 Tampere, Finland
Tel. +358 3 316 3376 Fax. +358 3 317 4102
joni.helin@vtt.fi

Keywords. generative programming, domain specific languages, template metaprogramming, data presentation
Classification. 6 month's work in 2nd year of post-graduate studies

1 INTRODUCTION

There is a vast number of systems and applications that have interactive parts that need to present the user with a view to some data and provide the means to manipulate it. This is often achieved with custom graphical user interfaces using various GUI widgets. However, this can easily become laborious not only on the part of the implementor but also on the part of the user.

Minor changes to the underlying data model can lead to drastic redesigns of the GUI and its interfaces to the rest of the application. This can become a major issue in development if changes are expected to be frequent. From the usability point of view, opportunities for exploiting structural similarities in the data are easily missed, resulting in highly specialized widget compositions. Presentation might not succeed in reflecting the semantics of the data, further confusing the user.

We therefore need a way to produce a consistent view of the data in question to the user and also lift the burden of implementing the view from the application developer. Generative programming and domain engineering [CZA00] gives us a means to reason about the problem in a generalized context and arms us with the knowledge of how to automate the construction of software artifacts satisfying a given set of specialized requirements in the particular domain.

In this paper we present a solution that allows the application developer to conveniently define a data model related to application-level concepts in a domain specific language which is then automatically translated into source code. The generated code is responsible for managing the state of the data model instance and providing a uniform graphical interface for representing and manipulating the data model instance. In addition, a mapping of the data model into programming language constructs is presented to allow compile-time typesafe access to and mutation of the data.

The rest of the paper is organized according to the process of generative programming [CZA00]. In section 2, the domain of interest is defined and requirements analysis and domain modeling is based on examining two sample applications and their data. Section 3 goes on to discuss the design and implementation of a generative domain model [CZA00] for defining and generating data models. Finally we conclude the paper with some remarks on the results of the approach in an industrial setting.

2 DOMAIN ANALYSIS

We begin by naming and scoping the domain of interest. As described earlier, we are dealing with the problem of data presentation, i.e. we need to visualize a given data model and its state (data model instance). We narrow our scope by focusing on hierarchical data organizations which may contain recursion. This excludes more complex data organizations, for example the relational paradigm, but suffices to serve many practical needs without excessive increase in complexity. It also allows for an intuitive strategy of mapping the data model to a graphical representation using trees.

By analyzing the needs of two planned commercial applications, namely a GUI-builder for mobile platforms and a software installation package creation tool, we were able to extract and generalize requirements for defining data models that needed to be represented. For example, a GUI-builder needs to allow the user to manipulate properties associated with various GUI widgets, such as name, dimension, location and various options. The hierarchical structure of a menu system can be described by a data model which allows the user of the GUI-builder to define appropriate menus and submenus by composing them from other menu elements. Also when creating an installation package, there has to be a way to define which files are to be included, which locales are supported, what are the various properties and capabilities of the application being deployed etc.

It is clear that many application-level concepts can be naturally represented by an invariable hierarchical structure which consists of named components. Hierarchical composition portrays semantic relationships between components, which can be exploited to create an abstraction hierarchy to ease understanding and navigation. Each component of the structure has an associated value that determines the state of the component. The structure and range of a value of a component depends on the concept it represents. We identified the following kinds of data structures needed to represent various values of components.

- scalar values (integral numbers, real numbers, boolean values, character strings etc.),
- symbolic enumeration constants naming certain data values,
- collections of homogenous data values with and without duplicates,
- grouping of heterogenous data values into fixed-size tuples, and
- selection from alternative data structures.

In addition, the applications required further abilities for constraining value ranges by giving predefined sets and intervals of values and by restricting the form that string representations of values are allowed to take. The nature of the applications also indicate a need to alter the state of a data model instance programmatically in addition to providing an interface to the user. This should preferably be accomplished in a type-safe way in the confines of a programming language to avoid run-time errors due to inconsistencies between the data model and the code accessing it.

Based on the requirements gathered in domain analysis we are able to formulate appropriate abstractions for describing the particular data models in our domain. The structure of our data models is described using hierarchic arrangements of named *attributes*, which are either *composite* or *atomic*. The former are used to group attributes together and the latter to carry actual values. Attribute values are instances of typed *data*, where the *type* of a particular data instance is invariable. Different data organizations and constraints on data values can be conveniently represented by a conventional type system with basic types that can be used to form more complex types using various type constructors.

As a part of domain modeling we must identify what are the similarities and what are the variations in our domain. Obviously, similarities are located in the abstractions described earlier. They form the framework for discussing variations between different data models of interest to application developers. In our case, the bulk of variation is found in the variability of structure, instead of the variability of functionality. This follows naturally because data models are inherently structural entities, consisting of various compositions of data

definitions. We can therefore conclude that conceptual modeling can be used to describe the overall structure of our domain. To derive a conceptual model of our particular type system, we can effectively utilize feature modeling [CZA00] by considering the gathered requirements for representing and constraining data values. Figure 1 shows a diagram of the feature model for our type abstraction.

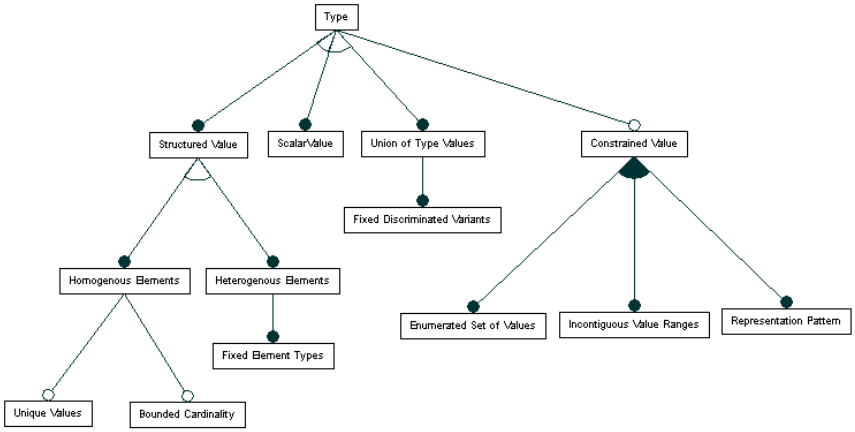

Figure 1. Feature diagram for the type concept.

3 DOMAIN DESIGN AND IMPLEMENTATION

We are now ready to move to the engineering part of domain engineering and design and implement a *generative domain model* [CZA00], which allows us to automatically generate a family member (i.e. a data model) based on a given specification. The architecture of our generative domain model is based on a meta model, which defines the expressible data models. The meta model, associated generators and other implementation components will be implemented in C++. Figure 2 depicts a partial class diagram of the meta model in a slightly adapted UML notation.

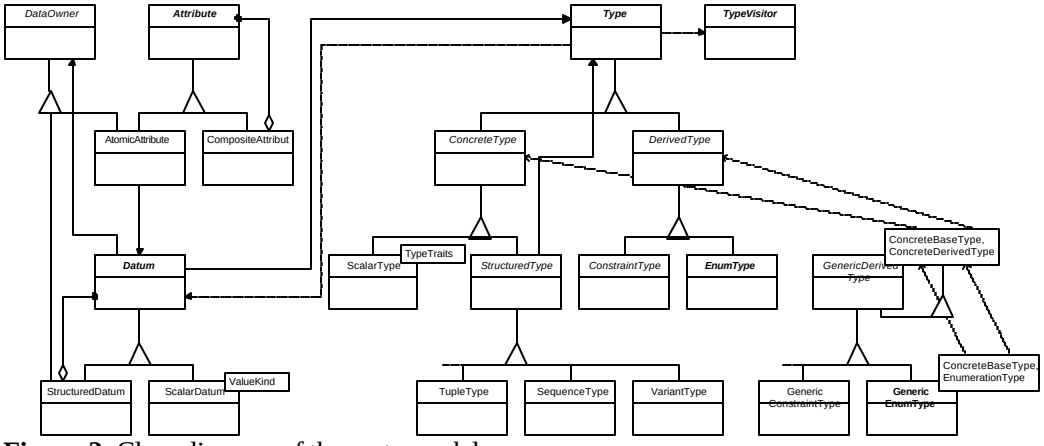

Figure 2. Class diagram of the meta model.

Meta model The abstract class Datum represents a value with a type. A Datum object offers services to access its type and owner, clone itself using the *Virtual Constructor* idiom [COP91] and notify interested listeners of state changes in accordance with the *Observer* pattern [GOF95]. The derived class StructuredDatum represents values that are made up of more elementary values using the *Composite* pattern [GOF95]. Derived class ScalarDatum represents a value that has indivisible state, parameterized by the type of the variable containing the value.

At the top of the type class hierarchy is an abstract class Type, which defines an interface for handling values of a particular type. The Type class supports accessing its name, creating values of its type, determining whether a value is an instance of the type, establishing equality and ordering between values and finding the successor of a value in order to iterate the value set. In order to keep the presentation terse, we elide descriptions of the more obvious type classes. It is to be noted that although the meta model permits (acyclic) sharing of data, such data models are not in our domain because the sharing is difficult to intuitively visualize by tree representations.

Constraining arbitrary existing types is achieved using template metaprogramming [CZA00], in particular parametrized inheritance where a given base type is derived to produce a new type with further restrictions on the values of the base type. In our type system we have several constraint type classes. For example, GenericEnumerationType can be used to select a set of values and label them with symbolic names by enumerating the values in question. GenericConstraintType allows for defining noncontiguous value ranges by giving non-empty intervals that define valid values of the resulting type. By combining various type constraints appropriately, syntactic restrictions can be easily enforced.

The meta model supports introspective processing of data models by providing an interface in the form of the *Visitor* pattern [GOF95, ALE01]. However, all the types to be visited are not known when the meta model is implemented. To circumvent the problem, we have arranged the composite types to be visitable in slices that correspond to the applied type constructors. Instead of offering a method per each concrete type composite, we provide methods for visiting the various kinds of parametrized type constructor classes. Parametrized inheritance chains are then flattened at runtime by requests to be dispatched up the chain one step at a time and thereby gathering type information accumulatively.

Graphical presentation of data models An important part of our approach is the graphical presentation of a data model instance to the user. A generic implementation is based on the introspection interface defined by the meta model. Multiple presentation schemes are possible, but an intuitive solution is to use a tree widget to display the structure and state of a data model instance and provide for editing the data values. Manipulations performed by the user are handled by the meta model instance corresponding to the data model, and changes that violate the type of a data value are flagged as erroneous. As an example, Figure 3 represents an instance of the data model description given in Figure 4.

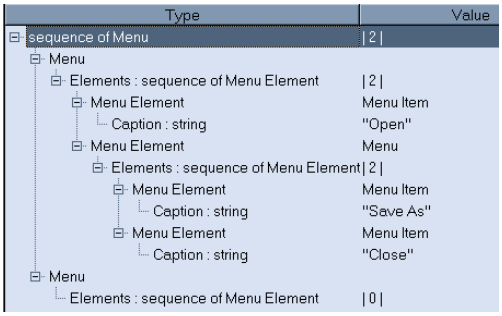

Figure 3. A tree representation of a menu system.

```
ATTRIBUTE 'Menu System' IS
  'Menu Element' IS VARIANT;
    'Menu Item' IS RECORD OF
      'Caption' : STRING;
    END;
    Menu IS RECORD OF
      Elements : SEQUENCE OF 'Menu
                 Element';
    END;
    'Menu Element' IS VARIANT OF
              (Menu, 'Menu Item');
    TYPE SEQUENCE OF Menu;
END
```

Figure 4. Description of a menu system.

Generator for meta model instances In addition to the meta model, we need a generator that is able to produce an instance of the meta model from a data model description. This greatly relieves the burden that would lie in the error-prone manual composition of meta model class instances. As noted before, variability is found in the (possibly recursive) structure of data models, for which template metaprogramming is not well suited. Therefore we define a context-free domain-specific language (DSL) for describing data models. A simplified extract in extended BNF is given below:

```
DataModel ::= "DATAMODEL" Identifier "IS" TypeDecl* Attribute* “END"
TypeDecl ::= Identifier "IS" (TypeDef | ForwardDecl) ";"
Attribute ::= CompositeAttribute | AtomicAttribute
CompositeAttribute ::= "COMPOSITE" Identifier "IS" TypeDecl*
                       Attribute* "END"
AtomicAttribute ::= "ATTRIBUTE" Identifier "IS" TypeDecl*
                    "TYPE" TypeDef ";" "END"
TypeDef ::= ( TypeReference | SequenceType | SetType | TupleType
            | VariantType | EnumerationType | PatternType | ...
            ) [ RangeConstraint ]
Value ::= IntegerValue | RealValue | BooleanValue | StringValue
        | SetValue | SequenceValue | RecordValue | VariantValue ...
```

We can implement a scheme to translate a description into C++ code that supports the generation of meta model instances corresponding to the attributes, types and values of the particular data model. An application can then call upon the generator to gain access to a data model instance. In our implementation, most of the semantic validation of the data model description is left for the C++ compiler, which is utilized to check for e.g., proper scoping, name clashes and restrictions on type compositions.

Supporting typesafe access A drawback of the meta model approach is that even though the application developer knows the structure of the data model and types of values, this knowledge can not be utilized to conveniently and safely access values contained in the data model instance. This is a direct result of offering a uniform interface to process data models, forcing the use of downcasts to access concrete Datum and Type objects. A solution is to define a mapping from the constructs in our meta model to constructs of the target programming language. In our scheme, attributes, tuples and variants map to (nested) structs, containers to vectors and scalars to appropriately typed values. Recursive types are mapped to pointers to respective type objects to prevent infinite recursion. The scheme results in a highly analogous structure with the data model which facilitates ease of use. To complete the mapping, we have to generate code for translating the state of the meta model instance to the corresponding mapped data model instance and for keeping them syncronized.

4 CONCLUSIONS

In this paper we have presented a way to apply generative programming in the domain of data presentation. As a result, the meta model and domain specific language for defining data models significantly alleviates work needed to support applications with the capability to provide the user with a graphical representation of the underlying data. In practice, it has been observed that the data model domain described in this paper is appropriate for catering to the requirements of a variety of applications. The concrete implementation of the approach has been componentized into a convenient solution that has been found easy to use and apply in industrial projects.

REFERENCES

[ALE01] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001

[COP91] Coplien, J.: *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991

[CZA00] Czarnecki, K.; Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Reading, 2000

[GOF95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Addison-Wesley, 1995