4. Tool encapsulation architecture

4.1 Overview

Principles 1 (“granularity of design objects”) and 2 (“design representation in files”) presented in Section 2.10 clearly deviate from the classical assumption that granularities of design data storage, browsing, and transport are equal. In this scenario, design data are either handled at a file-based granularity or at a value-based granularity. On the other hand, we want to achieve granularities as follows:

aspect of data handling	storage/ browsing (DIM)		transport (file system)		processing (tool)
granularity	object-based	**(1)**	file-based	**(2)**	value-based

Table 8. *Granularities for storage, transport, processing and browsing of design data*

While gap (2) is bridged by the input/output processors of design tools themselves, our encapsulation mechanism has to consider gap (1). As outlined in the Introduction, this encompasses

- the analysis of design files and storage of extracted structural information in the design information management component of the framework, and
- the construction of design files from structural information and design representation managed by the framework.

The structural information is managed separately from the design representation information which is stored as text chunks attached to design objects. On export, these text chunks have to be recombined as required by the current design step and design configuration to be fed into the design tool.

For control integration, our minimum requirement is that we have to be able to communicate an initial request for some activity from the framework to a design tool and to transfer notifications of activities on design objects between framework and tool. Despite this low level of control integration without intermediate requests being

passed to the tool, there can be no general solution due to a lack of agreed standards in the area of tool control.

Even less can be achieved for presentation integration. As tools generally do not provide access to their internal object structures, it is by and large not feasible to combine browsers that work on framework data structures as well as on tool data structures. Integration in this area has to be achieved on a case-by-case basis. At least, due to the possibility to manage meaningful relationships between design objects in the design information management (DIM) component, useful browsers can be implemented on top of it.

Principle 3 of Section 2.10 prescribes that design tools be encapsulated into the framework. Yet, with purely file-based storage and transport, writing the appropriate tool wrappers can be quite lengthy and error prone due to the lack of standard behaviour on the side of design tools. The amount of work would simply be overwhelming if we attempted to bridge the gap between object-based storage and file-based transport in an equally ad-hoc manner. A new framework service is called for. This service replaces the UNIX shell conventionally used for programming wrappers for encapsulated tools. By looking again at the execution protocols introduced in Section 2.8, we conclude that such a service must be able to perform the following tasks:

1. Determine affected design objects from tool arguments
2. Elaborate dynamic version bindings
3. Map between physical design files and text chunks attached to design objects in the design information manager
4. Check in/out design objects
5. Establish an execution context (file system directories, environment variables, global libraries, start-up files, resource files)
6. Start tool
7. Maintain message connection with activity in tool
8. Scrutinize tool execution

4.2 A new framework service

We assume that tight integration is too expensive to be feasible for a majority of design tools so that in many cases only encapsulation will be attempted. We assist design tool encapsulation by an architecture that directly supports the tasks

addressed by an environment builder with appropriate services. As a first step towards such an architecture, we refine the CFI reference architecture (cf. Figure 3 on page 15) as shown in Figure 7.

In this refined architecture, an encapsulated tool is augmented by a set of tool-specific adaptors. While the tool is not aware of its encapsulation into a framework, these adaptors connect to the public interfaces offered by the framework services in much the same way as tools that are directly integrated with these framework services would. From the framework point of view, an augmented tool behaves just like a tightly integrated tool. As suggested in this architecture, tool adaptors are considered tool specific and not subject to direct framework support. Thus, ad-hoc techniques prosper, giving little room to the reuse of adaptor code.

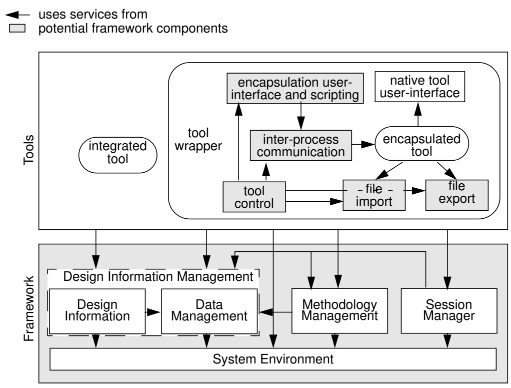

Figure 7. *Service view of the CFI reference architecture. Also shown is an encapsulated design tool with its tool wrapper that performs file import and export and some other services.*

Our approach is to consider the adaptors needed for tool encapsulation as additional framework services. Code can be shared among tools with similar adaptor requirements. For example, all tools that interface to the design description language VHDL can reuse the same file processing adaptors. Encapsulated simulators can share the same inter-process link to a text editor that displays the source line currently executed. We conclude that adaptors for encapsulated tools should be part of the framework (Figure 8). While Figure 8 identifies the tasks addressed by the new tool encapsulation service, we will introduce a detailed component architecture in the next section.

4.3 Component architecture

4.3.1 Language selection

The new encapsulation service is built as a language driven, programmable engine. This architecture provides the required flexibility to cope with widely different encapsulation tasks. At the heart of this engine is an interpreter of the Tool Command Language *Tcl* [Ousterhout 91].

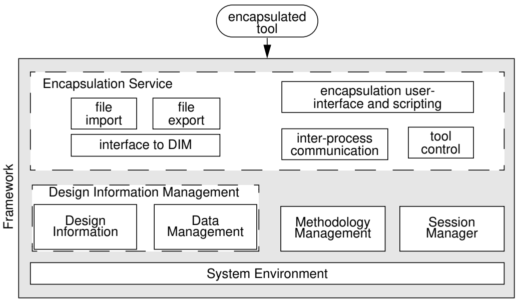

Figure 8. *Service view of the framework, extended with tool encapsulation service*

The main reason to select Tcl over CFI's choice *Scheme* ([CFI-EL 91]) is its modelessness:

> *"Tcl does not enforce any particular programming paradigm such as functional, logic, or object-oriented."*
> *[Brannon 94], p. 12*

The language is not to replace the framework's extension language but to supplement it with a language that is more familiar to tool integrators already being exposed to UNIX shell programming. Tcl was chosen over its competitors for the following reasons:*

- The language syntax is close enough to UNIX shell languages so that tool integrators, familiar with writing shell wrappers, will quickly adapt to the new language.
- It is interpreted, so tool integrators and even designers can readily configure the design environment. This is especially useful when encapsulating new design tools or extending the granularity at which design data is managed by the design management system. Not least, it provides an easy-to-use environment for rapid prototyping of new design management tasks.
- It has but one data type, i.e. character strings. Experience shows that what looks like a serious flaw both for script writers and from the standpoint of efficiency, hardly ever is a restriction. However, it dramatically simplifies the interface to the language from "C" as well as the language itself. When more complex types are needed, time-critical parts of an encapsulation wrapper can always be written in "C". The "C" code is demand-loaded into the engine and becomes available as new Tcl commands. From Tcl scripts there is no visible difference except for improved execution speed.
- It has a complete language kernel with variables, lists, arrays, procedures, as well as string and file handling capabilities. An object-oriented extension is available to help structuring large applications.
- It is designed to be embedded in applications. It has an orthogonal, well designed and documented procedural interface that makes it easy to define new commands, access variables and other state information in an interpreter.

*. Despite this list of good arguments in favour of Tcl, our new encapsulation framework service could have been based on Scheme, with some more effort needed for the implementation and adaptation to a lisp-like language for the tool encapsulator.

- It is easily extendible. A single procedure call in "C" is required to register a new command with an interpreter.
- It is integrated with a powerful and easy-to-use, Motif compliant toolkit for the X11 Window System as well as for Microsoft Windows, so creating execution protocols which require fancy user interaction is possible not depending on a single platform without too much effort. The key features of this toolkit are a constraint driven geometry management and useful default values for all widget parameters so only the really necessary values have to be given.
- A versatile extension *expect* exists that allows to control a design tool via its input/output.
- Extensions exist which allow to easily use mechanisms for inter-process communication (IPC) such as sockets, remote procedure calls, or ToolTalk [ToolTalk].

4.3.2 The Tool Command Language

For the purposes of this thesis we will explain only that part of Tcl's syntax and semantics that is necessary to understand the Tcl code examples given in the sequel. For a more thorough introduction to Tcl and the Tk toolkit the reader is referred to [Ousterhout 94]. A Tcl command consists of a command name and a list of command arguments, separated by white space. Newline characters or semicolons are used as command separators. Each Tcl command returns a string result and a code signalling success or failure.

Four additional syntactic constructs give the language a Lisp-like character:

- Curly braces act as nestable quote characters used to group complex arguments.
- Square brackets are used to invoke command substitution.
- Dollar signs are used for variable substitution.
- Backslashes may be used to insert special characters into arguments.

There is only one data type in Tcl: *Strings*. ASCII strings are used for commands, command arguments, results returned by commands, and variable values. Some commands expect one or more of their arguments to have one of the special forms of string: list, expression, or command. What makes Tcl extremely usable in our

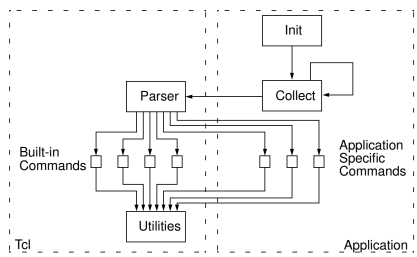

Figure 9. *Structure of an application that uses a Tcl interpreter [Ousterhout 90].*

context is that it is easy to integrate into applications. The interpreter consists of a small library of "C" functions used to invoke Tcl commands, to bind "C" functions defined in the application to new Tcl commands, and to manipulate the values of Tcl variables (Figure 9).

We have extended the interpreter with new commands to fit it to its new task as execution protocol engine. Key to the extension is a module we added to make compiled object-oriented classes, written in "C" or "C++", accessible from Tcl scripts. Objects can thus be created, manipulated and deleted. The public features of compiled classes have to be registered to the Tcl interpreter. This is achieved either by invoking a schema programming interface or by feeding the interpreter a schema via extended Tcl scripts. The wrapper facility interfaces to objects through a small set of well-defined procedures which have to be implemented for each public member of a class. We have implemented a generator that wraps the more regular cases of public data members and member functions, constructors, and destructors of "C++" classes.

It is desirable to load the parsers for particular design description language into the encapsulation engine on demand. Thus, support for new languages can be added or existing modules replaced when requirements change. We have therefore added a facility that is able to load a module with compiled class definitions into a running encapsulation engine. Demand loading can be triggered while loading a schema for a compiled class into the Tcl interpreter.

Other than the generic modules for the wrapping of compiled classes and for demand loading, the encapsulation engine is equipped with modules more specific to the task of tool encapsulation (cf. Figure 10):

- *Language independent parse tree and symbol table.* This module contains classes to construct and manipulate syntax trees and symbol tables in-core. A corresponding schema is fed to the encapsulation engine through the Tcl interpreter to wrap these classes and make them accessible from execution protocols.
- *The mapping between DIM Application Programming Interface (API) and the base framework API* is realized as a Tcl script and is loaded into the encapsulation engine at start-up time.

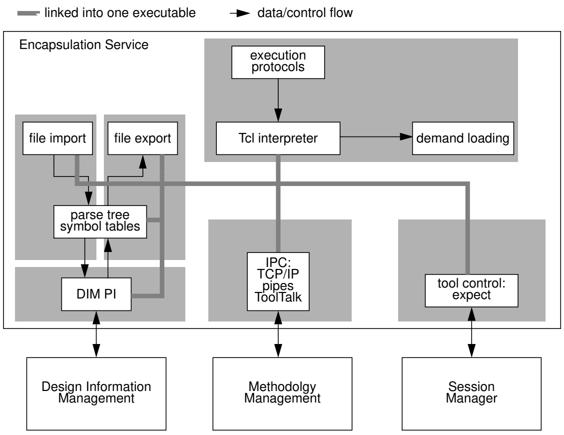

Figure 10. *Component architecture of the encapsulation service (services in Figure 8 are shaded)*

- *File import and export.* Whenever a new language needs to be supported, a new language parser module is generated from a specification and dynamically loaded into the engine. In addition, reading out information from the parse tree and storing it in the DIM system on import is performed in a Tcl script that is specific to a design description language. Export can completely rely on the information stored in DIM and is therefore carried out by standard methods, invariant to design language and tool changes.
- *Execution protocols.* Assuming that language parsers for the supported design description languages exist, the tool integrator is mostly concerned with writing execution protocols for the design tools to be encapsulated. Execution protocols handle all the details of preparing a tool run, starting and controlling the tool, and finally cleaning up and evaluating the results.

4.4 Wrapping compiled "C" and "C++" objects

The execution protocol engine needs to make a number of compiled modules accessible to the tool integrator. These include the procedural interface to the underlying DIM component, the scanner and parser for design description languages generated by scanner and parser generator tools, and the implementation of parse trees. Other modules, e.g. to support designer interaction, are conceivable. Although these modules may not be implemented in an object-oriented language like "C++", the object-oriented paradigm is well accepted and so even pure "C" implementations offer opaque handles as object identifiers. We have therefore chosen to uniformly wrap compiled modules with object-oriented classes, regardless of whether the underlying implementation is fully object-oriented or only object-based. A basic requirement for the wrapping mechanism was that it should blend nicely with the object-oriented extension *iTcl* [McLennan 92] we have adopted for our extended Tcl interpreter.

4.4.1 Schema representation

The challenge for the wrapping mechanism is to map between object identifiers in the compiled module and strings in the extended Tcl interpreter. As objects may be created both from compiled code and from execution protocols, the strings representing object identifiers need to be dynamically created whenever a new object is encountered. Figure 11 shows the information model we use to manage the schema

of a compiled module. As it is used to represent schema information, it can be regarded as a meta schema for compiled modules.

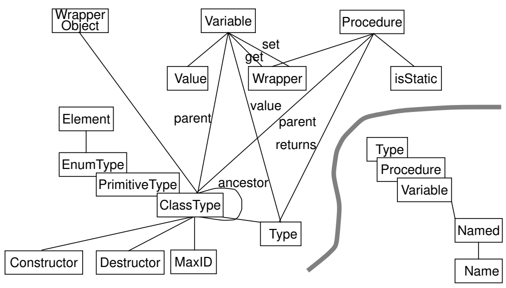

Figure 11. *Schema for the object wrapping module. The name attributes of* Type*,* Procedure*, and* Variable *are inherited from a common supertype* Named.

Every compiled object we want to handle from an execution protocol is associated with an object of type *WrapperObject*. A *WrapperObject* has a unique *Handle* and identifies its associated compiled object. *Identifiers* can be in-core memory addresses for "C" or "C++" objects, or database identifiers for objects residing in an (object-oriented) database*. *WrapperObjects* are typed, that is, they are associated with a *ClassType* object.

As subtype of type *Type*, *ClassTypes* are named. In addition, they may have a number of data members (objects of type *Variable*) and member functions (objects of type *Procedure*). *ClassTypes* may also be involved in multiple inheritance by maintaining a list of *ancestor-ClassTypes*.

Variables and *Procedures* both have a *Name* and (value- or return-) *Type*, which can be either a *ClassType*, *EnumerationType*, or one of the *PrimitiveTypes* void, integer, boolean, float, or string.

*. The types *Handle* and *Identifier* are not depicted in the schema diagram because they have a one-to-one relationship with type *WrapperObject*.

WrapperObjects access data members and member functions of their associated wrapped objects via wrapper procedures written in "C". Data members are wrapped by associating them with a pair of wrapper procedures for read and write access. The *set-Wrapper* procedure may be omitted for read-only data members. Two different signatures are defined for these wrapper procedures:

```
typedef int (*Enc_Wrapper)
  (Enc_Info* info, Enc_Identifier obj, Enc_Value& result,
   int parmCnt, Enc_Variable parm[]);
```

Procedures of type *Enc_Wrapper* are used to wrap ordinary member functions or give read or write access to a data member. An object identifier is required.

```
typedef int (*Enc_StaticWrapper)
  (Enc_Info* info, Enc_Value& result, int parmCnt, Enc_Variable parm[]);
```

Procedures of type *Enc_StaticWrapper* are used to wrap static member functions or constructors where no object identifier is needed as a parameter.

Using a purely procedural interface to wrapped objects has the advantage that even objects implemented in "C" may be wrapped. Only some kind of opaque handle mechanism is necessary which is a common concept in both operating system libraries (e.g. FILE*), and design data management systems (e.g. CELL_HANDLE, STREAM_HANDLE in Nelsis).

The implementation of this meta-schema facility provides a set of classes and member functions to easily define *Types* and associate them with *Variables* and *Procedures*. At run-time, each *ClassType* object maintains its extent in a hash-table to efficiently map from handles to object identifiers. The reverse direction is also supported: Whenever a new object identifier is encountered, maybe as a result of a procedure call or as value of a data member, that has not yet been assigned a handle, a new object handle is generated consisting of the class name and a unique number.

As the whole system is not persistent, object handles are only valid within a single session and are not suitable as object identifiers per se. It is, however, possible to override the standard handle generation procedure and use e.g. a persistent object id as handle. This is achieved by writing a new procedure

```
char* genId (Enc_Type* type, void* obj);
```

and registering it with the schema programming interface. Thus, handles may be made valid between sessions or among different instances of the encapsulation service and can be used directly as object identifiers in inter-tool messaging.

Now that we are able to describe types implemented in compiled modules, the next step is to be able to load such modules into a running execution protocol engine. New modules, e.g. for new or modified design description languages, can be added to the engine by the tool integrator on demand. No relinking of the engine is needed. The implementation relies on the UNIX library functions *dlopen*, *dlsym*, and *dlclose* to implement this feature. The only requirement on the demand loaded module is that it must contain position independent code. This can be achieved with a compiler switch.

Using a procedural interface to define a schema has little overhead but is complicated to use and difficult to maintain. We have therefore defined two new Tcl commands that allow to write schemas in Tcl. The commands translate the class definitions read from a schema definition file into procedure calls to the meta-schema facility. They also automate the demand-loading of compiled modules into the execution protocol engine. Here is the grammar for the schema definition language, using the syntax notation introduced in Section 3.2:

```
rule tcl_command {
   tcl_builtin_command
  | “enc_enum” IDENTIFIER “{“ repeat { STRING_LITERAL } “}”
  | “enc_class” IDENTIFIER opt { module:IDENTIFIER }
    “{”  opt { “inherit” repeat { class:IDENTIFIER } }
        list { class_member }
    “}”
}
rule class_member {
   “constructor” c_proc
  | “destructor” c_proc
  | { “method” | “proc” } type_designator IDENTIFIER
    “{” list { variable } “}” c_proc
  | “public” type_designator IDENTIFIER get:c_proc opt { set:c_proc }
}
rule type_designator {
   “void” | “int” | “float” | “boolean” | “string” | type:IDENTIFIER
}
```

What is most remarkable about this schema language is the handling of demand-loadable compiled modules as well as the use of “C” function names to associate procedures and data members with the appropriate wrapper functions. To automatically load a compiled module, a class definition may optionally name the location of this module in the UNIX file system. A table of already loaded modules is maintained so that a module is loaded only once. When the module name is omitted, the corresponding class definitions are assumed to be compiled into the engine. Naming “C” functions works in both cases. Whereas only fixed numbers of positional parameters are supported for ordinary methods, constructors and destructors are allowed to have arbitrary, named parameters. The schema programming interface allows both alternatives for either type of function.

4.4.2 Object use

Once the engine has a description of the classes defined in a compiled module, objects can be created from classes that define a constructor. In the interpreter into which the schema has been read, each class defines a new Tcl command named after the class. New commands are also created for static member functions, named with class name and member name separated by “::”. The class commands are used to create new instances of the associated compiled objects with the following basic command syntax:

```
rule class_command {
   class:IDENTIFIER { object:IDENTIFER | “#auto” }
   list { named_parameter }
  | class:IDENTIFIER “info” { “inherit” | “heritage” }
}
rule named_parameter {
   “-” parameter:IDENTIFER* typed_value
}
rule typed_value {
   INTEGER_LITERAL | FLOAT_LITERAL
  | BOOLEAN_LITERAL | STRING_LITERAL
  | enum:STRING_LITERAL | OBJECT_HANDLE }
}
```

The result of executing a class command is a new object handle, either automatically generated or chosen by the programmer. In addition a new command is created in the Tcl interpreter that is used to invoke methods on the new object. A class

[*]. no space is allowed between the dash “-” and the parameter:IDENTIFIER

command may fail if the underlying constructor function signals an error, e.g. because of conflicting lock requests. It is the responsibility of the wrapper function for the constructor to check the validity of any named parameters passed on the command line. A class command may also be used to retrieve the direct ancestors of the associated class from the schema data structure.

The purpose of invoking a constructor through a class command is not always to create an entirely new object. As the execution protocol engine maintains no persistent memory in itself, a constructor call is always needed to pull an object from a compiled module into the realm of execution protocols. In these cases the named parameters allowed on the constructor command line may pass selection criteria like names to the constructor implementation. The check-out of a design object in Nelsis could be regarded as a typical example of this. The Nelsis DMI returns a CELL_HANDLE which is used as an object identifier in the wrapper module. The destruction of the wrapped object using the standard message *destroy* on an object handle may then actually check in the design object. This application shows why destructors may have parameters in execution protocols, too. Nelsis requires a completion code to accompany the check-in operation.

Object handles need not be created with class commands. They can either be returned from procedures or be referenced through a data member of a wrapped object. The object handles are created the first time the procedure is invoked or the data member is accessed. The object handle can be used as value wherever an object of that type is expected. In any case, its value is also the name of a new Tcl command that is used to invoke methods on the associated object:

```
rule object_command {
   object:IDENTIFIER “destroy” list { named_parameter }
  | object:IDENTIFIER opt { ancestor:IDENTIFIER “::”* }
   method:IDENTIFIER list { positional_parameter }
  | object:IDENTIFIER “config”
   repeat { “-” attribute:IDENTIFIER* typed_value }
  | object:IDENTIFIER “info”
    { “class” | “inherit” | “heritage” | “proc” | “method” | “public” }
  | object:IDENTIFIER “isa” class:IDENTIFIER
}
rule positional_parameter { typed_value }
```

[*]. no space is allowed between the ancestor class name, the two colons “::”, and the method name

Methods and data members are searched in the class of the object first and then, if not found, recursively upwards the inheritance hierarchy of the object class. A list of class names in exactly the sequence of method lookup can be retrieved by issuing the request *info heritage*. This default search can be overridden by prefixing a method name with the base class that defines the requested method. Data members can be set with the standard message *config*. A method with the name of the data member is automatically provided to read the value of a data member. Other standard messages exist to query the type data structures for certain information. The standard message *destroy* invokes the destructor with optional parameters. A short example illustrates the use of class commands and the manipulation of objects:

```
/* file: class.h */
struct base;
struct derived;
struct base {
  int a;
  char* s;
  base* obj;
  base(): a(0), s(0), obj(0)      { printf (“base (0)\n”); }
  base (char* s_): a(0), s(strdup(s_)), obj(0)
                                  { printf (“base (s=’%s’)\n”, s); }
  int incr (int i)                { printf (“incr(%d)->%d\n”, i, a+i);
                                    return a += i; }
};
struct derived: public base {
  int b;
  char* t;
  derived(): b(0), t(0)           { printf (“derived (0)\n”); }
  derived (char* s_): b(0), t(0), derived (s_)
                                  { printf (“derived (s=’%s’)\n”, s); }
};
int add (base* a, base* b)      { return a->a + b->a; }
```

```
/* file class.tcl */
enc_enum blubber { blubber waber glibber }
enc_class base libclass.so.1.0 {
 constructor                                   new_base
 destructor                                    del_base
 method int incr { {int incr} }                base_incr
 public int a           base_get_a             base_set_a
 public blubber soft    base_get_a             base_set_a
 public string s        base_get_s             base_set_s
 public base obj        base_get_obj           base_set_obj
}
enc_class derived libclass.so.1.0 {
 inherit base
 constructor                                   new_bar
 destructor                                    del_bar
 public int b           derived_get_b          derived_set_b
 public string t        derived_get_t          derived_set_t
}
enc_class ex libclass.so.1.0 {
 proc int add { {base a} {base b} }            ex_add
}
```

Now we can create some objects:

```
% base b -s “aBase”          -> base (s=”aBase”)
% derived d -s “aDerived”    -> base (s=”aDerived”)
                                derived (s=”aDerived”)
% b config -soft glibber; d config -a 40
% ex::add b d                -> 42
```

The last line shows the use of a static member function. Please note that enumeration values are quietly coerced into integers.*

*. The result of this addition operation is of course purely coincidental and has no philosophical meaning whatsoever [Adams 79].

4.4.3 An example: The Nelsis DMI

The following is an excerpt from a schema definition for the Nelsis DDM system:

```
enc_enum completionModeEnum {quit complete}
enc_enum openModeEnum {read write update}
enc_enum purposeEnum {edit import derive export extract view}
enc_class DesignObject {}
enc_class Library $DDMLib/schemas/libnelsis.so.4.3 {
 constructor                                   new_Library
 destructor                                    del_Library
 method string query {{string query}}          Library_query
 method DesignObject designObject {
      {purposeEnum purpose} {string name} {string viewType}} \
                                               Library_designObject
 ...
}
enc_class DesignObject $DDMLib/schemas/libnelsis.so.4.3 {
 destructor                                    del_DesignObject
 public Library lib                            DesignObject_getLib
 public string mod_name                        DesignObject_getModName
 public int v_number                           DesignObject_getVNumber
 public string view_type                       DesignObject_getViewType
 public string hierarchy                       DesignObject_getHierarchy \
                                               DesignObject_setHierarchy
 ...
}
enc_class Nelsis $DDMLib/schemas/libnelsis.so.4.3 {
  proc void init {{string tool}}               Nelsis_init
  proc void quit {}                            Nelsis_quit
  proc void checkInAll {}                      Nelsis_checkInAll
 ...
}
```

Only some of the class members are shown here to demonstrate the use of the schema definition language. The data member *hierarchy* of class DesignObject is worth noting. Nelsis allows to access hierarchical relationships through the use of streams. The two wrapper functions *DesignObject_getHierarchy* and

DesignObject_setHierarchy map this stream interface to an interface to a pseudo data member *hierarchy*. Whenever there is read access to this data member, the corresponding stream is read and its contents is returned as value of this data member. On write access, the string passed as parameter to the *set* procedure is written as new contents into the *hierarchy* stream. The class *Nelsis* collects functions that are not associated with any particular class. As such its member functions are static and do not pass an object identifier as parameter. A typical use of these classes in an execution protocol would look like this:

```
Nelsis::init browser_tools
Library lib -path ~/projects/DP32 -mode update
browse “DP32”
lib destroy -mode complete
Nelsis::quit

proc browse {mod_name {lv 0}} {
 puts “[incr lv]: mod_name”
 set q [lib query “GET DesignObject
       WHERE     Module ITS Name == $mod_name AND
                 Module ITS ViewType == ’structure’”]
 if { [llength $q] == 1 } return
 set do [lindex $q 1]
 foreach h [$do hierarchy] { browse $h $lv }
}
```

This short Tcl program will open the Nelsis project “~/project/DP32” for update, and list the elements of the hierarchical composition of a *Module* “DP32”. All the real work is done by the procedure browse, which prints the *Module* name passed as parameter and then searches for a *DesignObject* with *Module* name “DP32” and *ViewType* “structure” in the Nelsis database. The method *query* of class *Library* provides the interface to the Nelsis DML. It allows to pass queries as strings and returns query results as a Tcl list that are easily processed by Tcl commands. The first list element always contains the list of attributes requested, followed by a - possibly empty - list of results. The first *DesignObject* found is extracted from the list. A loop then iterates over its children in the composition hierarchy returned as value from the pseudo data member *hierarchy*, calling itself recursively with the *Module* name of that child.