Storage mechanism for VHDL intermediate form

Bruno Poterie

Swedish Institute of Microelectronics

1. Definitions and Interfaces

1.1. IDL

An IDL model consists of a set of hierarchically defined classes, whose attributes may be numeric or symbolic constants, references to (instances of) another class, or a sequence of those. An instance of a terminal (leaf) class is called a node. Nodes belonging to the same IDL model, and referencing each other, form a cluster, which is the unit of transaction to and from disk. In addition, reference-type attributes may be global, i.e. between two different clusters. Finally, attributes may have the same name across separate class trees, provided that they are of the same type.

A given IDL schema also defines an associated library of functions which access and update individual attributes. These may be automatically generated, along with complete create-fill-node routines, print-out and scanning primitives to implement a browser. These functions form the attribute level of the SPI interface (see below). Note that some attributes may be hidden to direct access and/or to create/browsing.

1.2. SPI

SPI is a collection of definitions and functions, which allow a standard interface to the actual storage system. It may be divided into three layers:

- the top-level: from database root to cluster level
- the node level: creation of and access to nodes and sequences of nodes
- the attribute level: getting and setting the contents of a node

In our system, most of the top-level semantics are handled by a "VHDL Environment", VE, which through an interactive user interface deals with design data and library management, invocation of individual tools, storage of source texts, handling of versions, automatic dependency list retrieval and invalidation, thus ensuring a common interface and a coherent development and simulation environment. As a consequence, the SPI top-level layer contains itself with choice-Free management actions that tools need to get to the clusters:

- opening and closing the database
- searching and opening models
- searching, opening and closing libraries
- searching, creating, opening and closing units

INTRODUCTION

VHDL (Ref. 1) analysis is a complex and time consuming task. It is therefore desirable to only perform it once, and then to use a static error-free form for subsequent references by other programs. This calls for long-term storage of a structured intermediate representation, hereafter called VIF (VHDL Intermediate Form). It is also desirable to insulate the applications from the actual storage implementation, thus providing safety and controlled data access. Therefore all data access is via a Software Procedural Interface (SPI).

In the VHDL system being developed at the Swedish Institute of Microelectronics (Ref. 2, 3), we decided very early to implement VIF using a private storage system, rather than using an existing database. Speed and memory requirements were predominant considerations here, since VHDL applications require large and (rather) infrequent transactions, whereas typical DB systems are designed for small and frequent transactions.

In order to allow constant development of the data schema and application of the software to other areas, both VIF and SPI are automatically generated from a data description written in the IDL language (Interface Description Language) (Ref. 4). The underlying ideas are inspired by those presented by CLSI (Ref. 5).
The second layer deals at the node level. It provides:

- testing a reference against NULL
- creating/deleting a node of a given class
- creating/deleting a sequence of references to nodes of a given class
- adding/removing a node reference to/from a sequence
- getting the size of/traveling through a sequence

The third level deals with the attributes, and provides set and get functions for each of them, together with create/fit and browse/printout functions for each terminal class (see above).

2. Rationale for the Implementation

2.1. Criteria

Apart from the obvious criteria (fast, compact, simple, reliable), the following requirements remained constant guidelines:

- keep the various parts (file storage, internal structures, memory management, SPI layer) as independent as possible.
- try to hide as much as possible from the final user, to avoid having to update user programs after each internal change.
- keep actions small and stupid (no "clever" side-effects, the user has to specify what he wants).

2.2. Not a real database

The decision was taken to implement a private storage system, for performance reasons. Basically, we wanted a system whose time and memory consumption would remain proportional to the size of the VHDL source involved. Relational DBMS have a tendency to prefer a square ratio or worse. Other types of DBMS are not so widely distributed.

2.3. Parts

2.4. Unique ID

The three main packages (SPI, IN, VFS) share a common feature, the unique ID, which is a number/reference associated with each object, be it an IDL model, a library, a unit, or whatever. This ID is distributed inside the UNIVERSE, which is a host sub-directory, managed by and accessed through VFS. Application programs may specify which UNIVERSE is to be used when connecting themselves to the system.

2.5. Fixing references

One further important decision, motivated by performance requirements, was that once in the workspace, references between nodes would be replaced by actual pointers, which are typically followed more than once. Therefore the need of a "fixing" phase, between the two representations.

The drawback is that both forms may coexist inside a given workspace, depending on the order of opening of all necessary units. The application tool is responsible for loading all needed resources before accessing global references.

3. The VFS package

3.1. Goals

VFS stands for Virtual File System, and it provides a logical hierarchy of file access. All long-term storage is in fact implemented using one single host system directory, called "VFS", which VFS itself creates and maintains under the UNIVERSE. VFS provides facilities for creating, finding, updating, removing and locking models, libraries and units. It supplies the unique IDs which are then used by other components.

3.2. Objects and access

VFS knows about four main sorts of objects/files:

- HISTORY: contains, for each allocated unique ID, the ID of its owner, which is normally DIRECTORY, or may indicate TOP for root files, or CANCELLED if removed, or NIL if available again. The contents of the HISTORY may be read, but it can be modified by side-effects of other functions only.
XREFS: contains the list of logical dependancies between objects, as specified by the user. No assumption is made about the contents, except that it must refer to valid object IDS. Management is made by VFS for individual entries, or by the user globally.

DIRECTORIES: fully managed by VFS, each of them records the list of objects created directly under it, with name, creation date, last update date, unique ID, and a status word whose value is the sole responsibility of the user. The user may read the contents, find one entry by name, add one by name, and update the status words. The contents is otherwise modified by side-effect.

TERMINALS: data files, whose contents are completely unknown to VFS, which simply provides the access to the actual file. Access is typically write-one-read-many. VFS provides a twin temporary file for writing, which is renamed back to original on completion only.

Security is under the responsibility of the user, which can use the general GUARD semaphore and LOCKS on objects to implement the desired semantic. In addition, VFS will refuse to remove a Directory unless empty.

3.3. Actual interface

The underlying structures and functions allow for any hierarchy the user may wish to implement. Currently, the actual interface level implements a VHDL-oriented frozen model, which includes a root Directory for IDL models, and another one for libraries, which contains in its turn design unit clusters.

Still, this system is very general, and has in fact been used to maintain inside the same UNIVERSE several IDL models, clusters created and managed through programs hand-written directly on top of the IN interface, and even plain texts, together with a full set of VHDL libraries and units. Parallel access has been successfully implemented.

4. The IN package

4.1. Goals

This package defines and implements the actual structures and functions needed to create, delete, append, remove, and transfer to and from disk IDL models, clusters and nodes. It also implements the very important "fixing" phase, which switches between unique references and data pointers. In other words, it is concerned with the contents of the terminal files stored and accessed through VFS.

Data created dynamically during a session, i.e. descriptors and data nodes, use memory chunks returned by the "Structured Memory Management" package. This allocates from large blocks, until re-initialisation when all blocks are freed at once. This handler can manage as many subspaces as needed by the user. It has been introduced to reduce the number of system calls, and to improve the locality of pointers, which in turn influences frequency and speed of swapping.

4.2. Models

The workspace manages an array of models, indexes being allocated by the IDL compiler, reflecting creation order in the reference universe. Each model is however independant of the others. In particular references cannot cross models, and each cluster is modelled by only one model.

A model is stored as an array of descriptors, one for each class of the corresponding IDL description, whose low-order bits are used as an index into that array. Each entry contains the class name and the index of its immediate ancestor, or an impossible value (the class nb) if none. This allows the reconstruction of a class hierarchy, and a test of membership of a node to a given class. There is also a status word, indicating among other things if this class is a terminal class or not, and if it is declared as an entry point.

The class descriptor also contains the size of the attributes' part of each node in this class, i.e. the size of the 'size' part, and a pointer to an array describing those slots whose contents is of variable size and/or needs fixing. This array is stored on disk immediately after the slots:

<table>
<thead>
<tr>
<th>Type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>local ref</td>
<td>&lt;nodeID&gt;</td>
</tr>
<tr>
<td>global ref</td>
<td>&lt;unitID&gt;&lt;nodeID&gt;</td>
</tr>
<tr>
<td>local list</td>
<td>&lt;nb&gt;&lt;nodeID&gt;...&lt;nodeID&gt;</td>
</tr>
<tr>
<td>global list</td>
<td>&lt;nb&gt;&lt;unitID&gt;&lt;nodeID&gt;...&lt;unitID&gt;&lt;nodeID&gt;</td>
</tr>
<tr>
<td>string</td>
<td>&lt;nb&gt;&lt;char&gt;...&lt;char&gt;</td>
</tr>
</tbody>
</table>

Finally, it points to dynamically allocated memory blocks, which contain all nodes belonging to this class. This allows pre-allocation of groups of nodes (the number depending on the current node usage), and simplify subsequent processing. The list of logically deleted nodes is also maintained here.

4.3. Clusters

The workspace maintains a set of clusters. Each cluster consists of:
a header, with an identification part: name, libraryID, unitID, modelID which is stored on disk; a status word, saying if the cluster is valid, has been locally fixed, has all of its global relations fixed, is updated.

an array of list heads, one for each class of the corresponding model, chaining the nodes per class & unit. They are traversed at disk transfert time, or when handling with the class as a whole.

da direct access vector, with one entry per node, indexed by the nodeID, which is unique inside each unit, directly pointing to each node in memory. This is used during the fixing phase for fast direct access.

4.4. Nodes

A node in the workspace can be decomposed into 3 parts:

1. The volatile part is not stored on disk. This part consists of links to the model and cluster (class index and next node pointer within this class), a status word for fixing purposes (local? global?), and a book, which provides a generic attribute mechanism via the SPI interface.

2. The permanent part of the head is the unique identification of the node, consisting of both the unit ID inside VFS and the node ID inside the unit.

3. The slots contain the attributes, i.e. the actual application data. An attribute can be a numeric, symbolic or binary value, a string pointer, a pointer to another node or a pointer to a list of nodes (managed by the LH package).

4.5. Fixing

Fixing consists of replacing data on disk which refers to another object by the direct address of the object in the workspace. The basic unit of information is:

<table>
<thead>
<tr>
<th>Global</th>
<th>&lt;unitID&gt; &lt;nodeID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>&lt;nodeID&gt;</td>
</tr>
</tbody>
</table>

This is stored on disk directly after the slots, alone or in a sequence. When it is read, it is copied into a chunk of memory obtained from yet another subspace of the "Structured Memory Management" package, whose address is then stored in the corresponding slot. After the cluster is fully read, each node is processed, and for each such slot, the location function is called.

This function takes in turn the couples (unitID,nodeID), searches for the cluster in the set maintained by the workspace, and uses the node ID as an index into the direct access vector, which contains the node address, set at loading or creation time. This address is then stored in the slot, or added to an LH list. If it is nil, or the cluster could not be found, then fixing is abandoned for this node, and the already fixed slots are returned to the intermediary status, which is saved during node fixing. Hopefully the application will later load the good unit, where fixing will then again reprocess non fixed nodes.

This procedure is simplified for local references, which form the majority of references in VIF. Indexing in the direct vector is made in the local cluster, and a nil answer is considered as an aborting error – it most likely indicates a problem while transferring the cluster from disk, or that the reference was invalid. Accordingly, local fixing is performed first, and is not undone in case of unfound global references.

The routine which writes a cluster back to disk checks each node for its fixing status, and uses one of two sub-routines depending on the form. However, updates made on non-fixed slots are likely to be erroneous!

5. The SPI package

5.1. Work Unit

The Work Unit is the unit (specified by the application) where all creation and removal of nodes takes place. There are two basic problems to be considered. The first occurs when reanalysing an already existing VHDL design unit. What to do with the previous version? We decided to group all actions into a specialised function, NewUnit(name). This always asks VFS for a new ID, which then becomes the working unit. On successful analysis, the working unit is closed and therefore written to the disk. The old unit, whose ID was kept, is then cancelled from the system.

The second problem is that of creating a work unit while its external name is unknown. In this case, the NewUnit() function uses a pseudo-ID reserved by the VFS interface for this purpose. An attempt to close this pseudo-ID would be interpreted as an error. However, the same function called again with the real name will cause the substitution in the workspace of all references to the pseudo-ID by that of the real one.

5.2. Resources

Another problem is that of automatic handling of resources, specifically between units. Due to the non-DB nature of our system, this was very difficult to achieve. Instead, we decided to let each tool be responsible for specifying and loading all the units referenced as resources. We provided for that purpose a data structure, namely two ID lists attached to each cluster, and two functions to add to and load the contents of those lists. One of the lists concerns the primary references, which may be updated together with the main unit, that is, opened in WRITE mode; the second contains secondary references, which may only be opened in READ mode. The two lists are stored in front of each cluster.

Note that the default resource STANDARD does not belong to those lists, as it is loaded directly by the tools. Note also that following a global reference whose unit was not specified as a resource, and consequently not loaded, will result in a run-time error, possibly a core-dump.

5.3. Closing

When closing a library, all units belonging to it are automatically closed as well. An exception is the working unit (belonging to the working library), which has to be closed explicitly.
The workspace may be closed in forget (0) mode, or in normal (1) mode. The forget mode basically reinitialises the workspace, without updating anything on disk. The normal mode closes the work unit if not already done, then closes all libraries, and hence all units. It then cleans the workspace. Closing does not disconnect from the VFS, so opening of libraries, models and units can proceed immediately. Alternatively, one may choose to connect to another universe. Finally, the workspace is removed when the tool task exits.

6. Results and Evolution

6.1. Performances

Alltogether, the system has proven very robust and reliable, while retaining a lot of flexibility, as we continuously modified the top-level semantics and the environment features.

We can claim that the objective of linear performances has been achieved, as shown in the data given below. We have both speed and disk use comparing very favorably with published results (Ref. 6-7), except for opening time where we have to load all data in memory first.

<table>
<thead>
<tr>
<th>name</th>
<th>lines1</th>
<th>lines2</th>
<th>lines/sec</th>
<th>lnelsec</th>
<th>storage</th>
<th>storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VFS+IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table has been constructed from both referenced articles. The last column shows our figures for the same tests on the same configuration (SUN 3/160). It must be noted that the sequential scan is by far the most relevant figure for our application (SPI), which makes almost no use of the other facilities.

Next, some figures for the VHDL analyser using VIF and SPI:

<table>
<thead>
<tr>
<th>unit size (lines)</th>
<th>cpu elapsed (sec)</th>
<th>storage (Kbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1.9</td>
<td>3.0</td>
</tr>
<tr>
<td>65</td>
<td>3.3</td>
<td>9.3</td>
</tr>
<tr>
<td>137</td>
<td>2.1</td>
<td>9.3</td>
</tr>
<tr>
<td>513</td>
<td>13.0</td>
<td>8.9</td>
</tr>
<tr>
<td>1576</td>
<td>44.0</td>
<td>8.9</td>
</tr>
<tr>
<td>10013</td>
<td>547.6</td>
<td>8.9</td>
</tr>
</tbody>
</table>

The figures in the table are average times measured from the shell on a SUN 3/160 with 8 MB of memory. It can be seen that the performance seems to increase with the size of the analysed source code, up to a large optimum plateau which we believe will only rarely be crossed. The apparent anomaly of the third line is explained by the fact that this unit is a package specification, for which the analyser has much less to create and lookup. All other units are architectures containing numerous components. Note also that the fixed overhead for any unit uses almost all the cpu for the first example. Finally, an initial universe uses around 11.9 Kbytes, and the standard library occupies some 43.3 Kbytes.

6.2. Possible performance improvements

- The biggest single gain would come from using macro processing to replace the get() and set() attribute function calls by inline code and removing error handling. The gain is estimated to be as much as 30% of the time spent outside VFS and i/o. This would however remove the most efficient safety net, and significantly increase the binary size.

- VFS could make use of the UNIX lock(J) to implement the guard and locks, instead of empty files as is now the case. The main problem here would be portability.

- Finally, VFS could be transformed into a server, keeping frequently used data in fast memory, and using shared memory to transfer information. This would remove most of the constant overhead of connecting to the universe, loading the model and the standard resources, thus very effectively speeding up the handling of small design-units. The restructuring involved and the availability of shared memory and sockets are the main obstacles here.

6.3. Possible safety improvements

- All SPI functions accessing nodes could check that the data pointed to by their parameter actually looks like a correct node. Accessing global attributes could first check the fixing status of the node. When writing a cluster back, we could check that all global references point to units belonging to the resource lists. This costs time, and does not improve binary sizes.

- Another idea is to keep both fixing forms in each slot, and to perform fixing on request only, loading resources as required. The overall speed should not decrease very much, but the memory requirements would almost double. This also contradicts our requirement for small and dumb basic actions.

Acknowledgments

Firstly, the author wishes to thanks Ian Hill for inspired technical leadership and thorough review of this paper. The initial structures were designed largely by Nils Strandberg. The current implementation and interface have been largely influenced by frequent interaction and discussions with Matt Altmäe and Mats Fredriksson.

References

2. "VINT - a VHDL Simulation System" by Matt Altmae EDAC '86
3. "A neural net based self organizing scheduling algorithm" by S. Hamami, A. Adam Postula EDAC '86
4. "A tutorial introduction to using IDL" by Warren, Kickenson & Woodgrass
5. SIGPLAN Notices, '87
7. "Benchmarking simple Database operations" in B. Kuhler, R.O.G. Cattell
8. Sun Microsystems, Inc
9. "A performance comparison of Object and Relational Databases using the SUN benchmark" by Joshua Dahl & Craig Mason
10. Ontologic, Inc June '88