7. Case Study: Design system for high-level synthesis

7.1 Overview

In this chapter we show how to construct a design environment using the encapsulation methodology presented in this thesis. The purpose of this case study is to show that with the new framework service for design tool encapsulation and a reasonable effort for wrapper writing a design environment can be constructed. The encapsulation service has to incorporate the following elements:

- a design information management system implemented on an existing CAD framework
- language processors generated from HDL specifications
- a flexible extension language

We show that such an environment provides added value to the designer when compared to simple a-priori tool encapsulation based on design files. Apart from the fact that such a design system was indeed constructed, we report on the size of the extension language scripts that had to be written to build this environment.

The design environment comprises synthesis and simulation tools for electronic design, starting with a behavioural design description on algorithmic level and resulting in a structural description on logic level. It can be extended with tools to

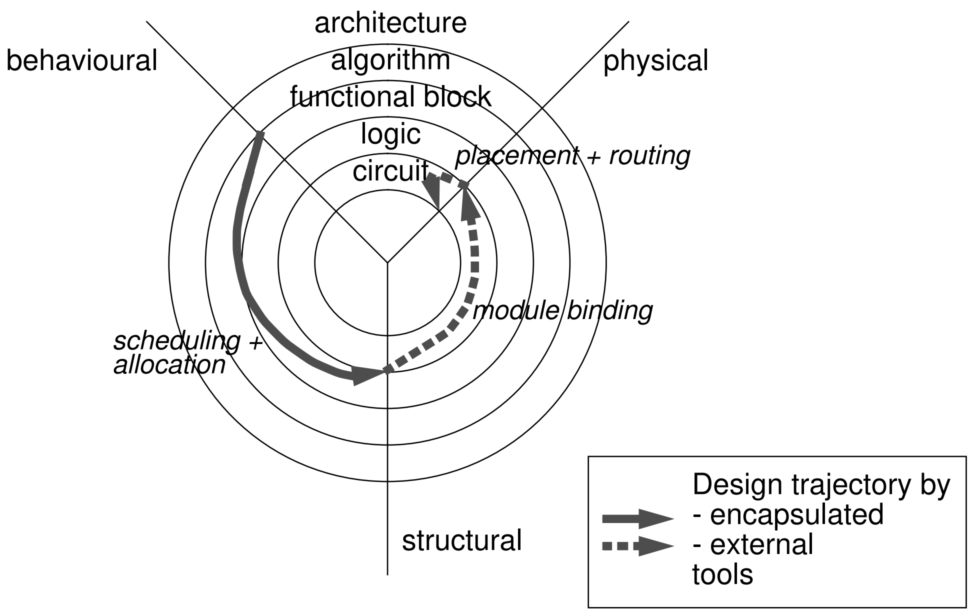


Figure 28. *Design steps possible with the prototype design environment*

transform the logic-level netlist description to a physical layout (Figure 28). Design information management is implemented using the Nelsis CAD Framework. The following design tools have been encapsulated:

- *Import*. Processes a set of VHDL files with behavioural or structural design descriptions and stores them as separate structural and representational information in the framework's design information management database.
- *Text editor.* The design environment supports multiple edit-compile-debug cycles which are typical for the design of complex systems. A text editor can be used to create an initial design description and feed it into the design information management system for administration. With the management of individual design units in the context of a comprehensive web of relationships, though, it seems more advantageous to create and edit design objects directly in the context of this web. Our implementation supports both approaches.
- *Synopsys design compiler.* The design compiler is a design environment in its own right, featuring a graphical user interface, a powerful scripting language and internal, binary design formats. It is able to read and write design descriptions in a multitude of formats for facets of a design like schematics, finite state machine descriptions, behavioural design descriptions, or structural netlists. Its purpose in our design environment is to synthesize a behavioural design description on algorithmic level, written in VHDL by a designer, into a structural design description on logic level, also described using VHDL.
- *Synopsys VHDL analyser.* The VHDL analyser is a batch tool. Its prime purpose in our design environment is to check VHDL files for syntactical errors and thus to guide the designer in constructing VHDL files acceptable to the other design tools and to language processors for design file export. Remember that we assume syntactically valid design files to greatly reduce the size of our language specifications. The VHDL analyser also produces an intermediate file with the analysis results that is needed by the VHDL debugger.
- *Synopsys VHDL debugger.* Much like debuggers for programming languages in software development, the VHDL debugger allows to analyse the runtime behaviour of a compiled design description. The VHDL debugger accepts hierarchical design descriptions on all levels of detail in all domains that are supported in VHDL. It allows to read stimuli, generate reports, single step through a design and examine the contents of variables and signals.

Simulation results are output in a proprietary waveform format that can be visualized by a separate tool.

- *Export*. Export takes a (hierarchical) design from the database and exports it to the file system as a set of VHDL files. This is needed because the design of an electronic system is not completed with the creation of a logic-level structural netlist. Further processing is necessary to transform this netlist into a physical mask layout with the aid of associated technology libraries. The physical mask layout can then be used as the basis for chip fabrication.

By using the Nelsis CAD Framework as the implementation platform for design information management, the designer can use the following framework tools provided by the Nelsis CAD framework as an addition:

- Graphical database query interface
- Version browser
- Equivalence browser
- Design flow browser

Note that using these framework tools only makes sense for a designer because it was possible to map DIM concepts naturally to Nelsis schema types (Figure 22 on page 93).

7.2 Design flow

The design environment arranges the encapsulated design tools in a design flow in which tools are linked to each other via typed data channels (Figures 30, 30). The overall structure reminds of a bus-based system in electronic designs in which design objects described in VHDL are transferred on the central channel or „data bus“. Design objects flowing between tools on this bus are managed by the design information management system. In addition, most of the tools can import or export design data external to the system. Some tools produce data in an opaque, binary format that, although being associated with a VHDL design object, has no internal structure that is known to the DIM service. The following subsections give a brief overview of the design tasks that are performed in a typical design situation using this design environment.

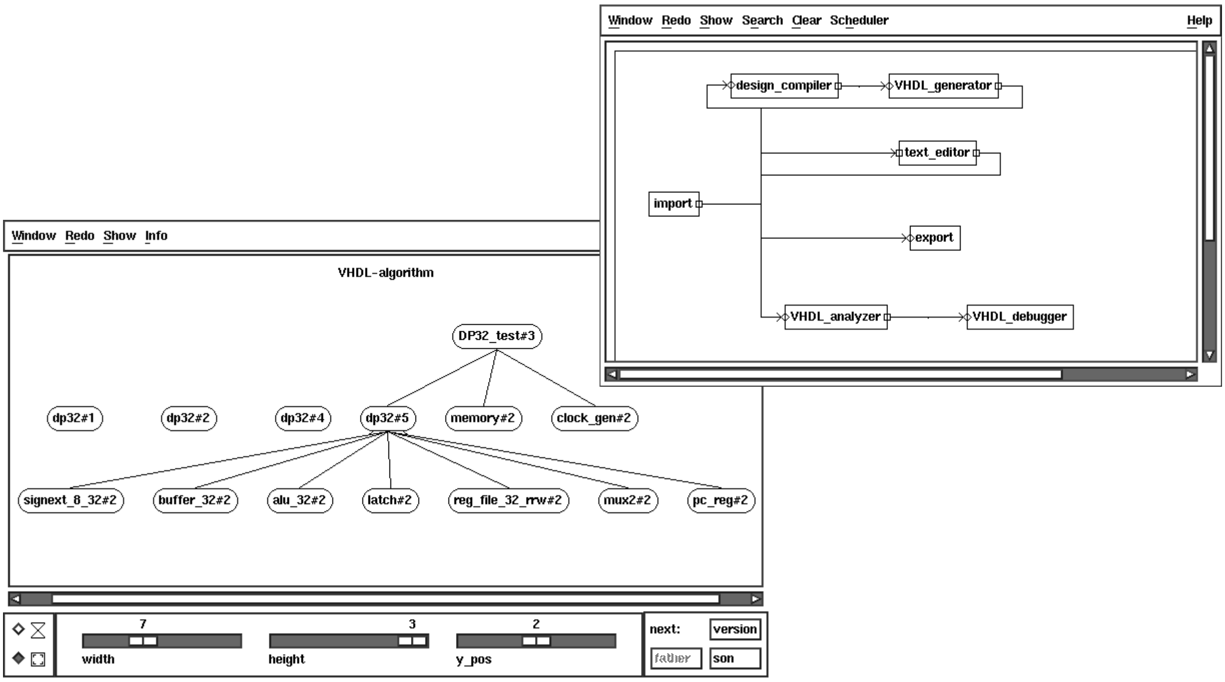


Figure 29. *Screen shot of a typical design situation showing the user interfaces of the two framework tools* hierarchy browser *and* flow browser *working on the DP32 processor.*

Creating and importing VHDL designs

A behavioural VHDL description on a low level of detail (typically algorithmic or functional block levels) of the design is created either manually using a text editor or a graphical specification tool, or as the output of some synthesis tool operating on even lower levels of detail. The design may be arranged in a single or in multiple design files and typically also references some external library elements. The VHDL files are imported into the DIM service, creating design objects for each entity declaration, architecture body, and configuration declaration, and establishing hierarchy relationships between them. The object graph thus created can be inspected with the hierarchy and version browsers provided by the underlying framework. Individual design objects can be viewed or edited by an encapsulated text editor which is a significant added value to the stand-alone tool suite.

Analysing design objects

The Synopsys VHDL debugger *vhdldbx* can be invoked on the top-level design object to validate the design. The debugger does not read VHDL objects directly but rather expects analysed **.sim* files produced by the VHDL analyser *gvan*. This design step reveals any syntax errors that remain in the design objects. If an error is encountered *gvan* pops up a window and allows to edit the design object.

Simulating design objects

After *gvan* has analysed the VHDL objects, *vhdldbx* can be invoked on the top-level or any intermediate-level design object. The design object hierarchy is traversed for this and exported as proper VHDL files into a temporary directory. Each VHDL file is accompanied by the intermediate file created by *gvan.* Within *vhdldbx* arbitrary simulation steps can be performed to validate the design.

Synthesizing design objects

Once the design is found to be correct it can be synthesized. Synopsys' design analyser is invoked for this task and fed with a set of VHDL files that have been extracted from the object hierarchy to be synthesized.

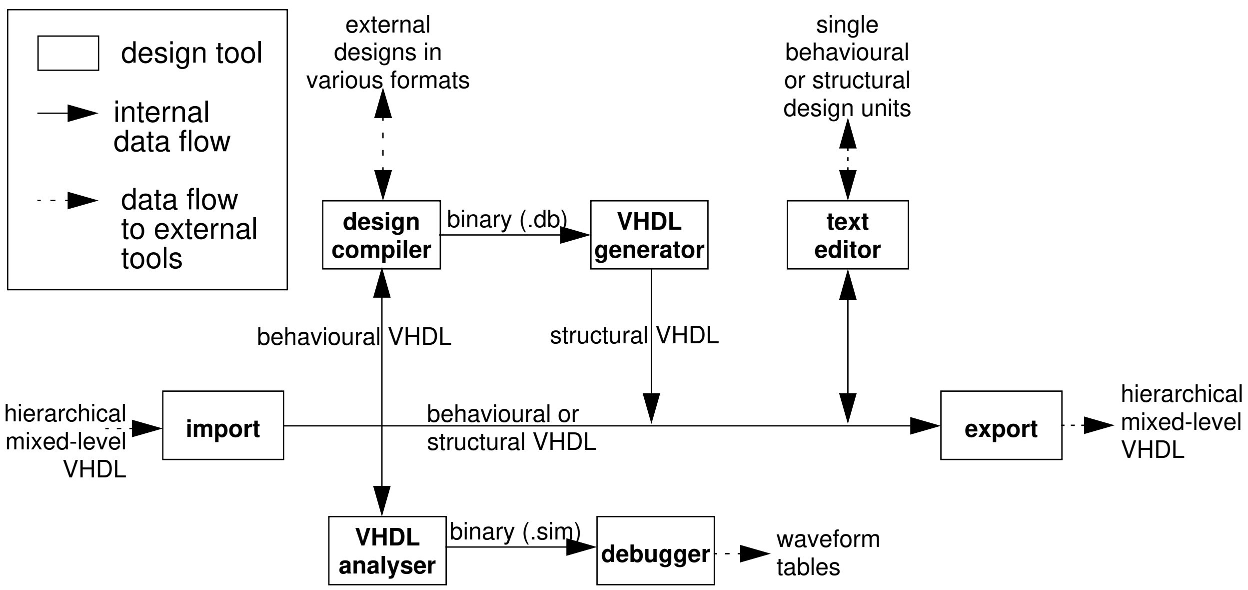


Figure 30. *Design flow. Dashed lines denote data channels which either enter or leave the environment and provide links for external processing.*

Deriving new alternatives

The synthesized, structural design is at first stored in intermediate **.db* files. This allows to perform an arbitrary number of synthesize steps until the result is satisfactory. An explicit design step *derives* a new VHDL description from these **.db* files by invoking the batch version of the design analyser *dc_shell* with an appropriate script in its native scripting language. Now the synthesized design objects may be inspected using either the text editor or the debugger.

Export

The final step in a typical design flow is to export the synthesized design into external VHDL files for further processing by external tools.

7.3 Building the environment

Building a design environment requires a number of tasks which have to be considered carefully. In the following, we will describe some of the decisions that lead us to constructing the design environment in the way we have.

7.3.1 Selection of the design tools to be integrated

The selection of the design tools to be integrated is probably the most important step because it determines to a large extent what the final design environment can be used for. It is not only important to select tools that offer a suitable functionality for a specific design task. The tools must also support a given design methodology. It is useful, for example, to already be able to simulate a behavioural description of a design and compare the result of this simulation with the results of a simulation conducted on some structural version of the same design. This implies that either a single simulator/debugger is available that can simulate both kinds of representation, or that two simulators are used which produce comparable results. We have chosen to base our design environment on a commercial suite of tools that contains a multi-level simulator which is capable to simulate both behavioural and structural VHDL files and to produce comparable results.

7.3.2 Dimensions of integration

A vital property of the selected tool suite is that the tools are *data-integrated*, that means that the output data of one tool can either be directly fed into another tool or that appropriate converters are available to convert the output of the first tool to a data format understood by the second tool. Encapsulated tools are *not* automatically data-integrated. In principle, our framework service with its toolkit for building parsers and unparsers can be used to construct format converters that read a design description in one description language, convert it into some language independent format and emit a description in another language. On the other hand, the granularity at which we manage design objects is too coarse to serve as intermediate, language independent design representation. This is of course exactly what we intended to keep language specifications small. The Synopsys design tools all support VHDL as a common data format. Some tools do not directly read or produce VHDL but need a pre- or post-processor to interface with VHDL. Further formats can be used by the tools but are not supported for integration purposes in our design environment. To summarize, we do not attempt to build any format converters in addition to the ones supplied with the tool suite.

Another desired property of a tool suite is that it is control-integrated. Control-integration means that tools can exchange messages and react to them in an asynchronous manner. Exchanging messages can be used in a design system in a multitude of ways. Some applications are:

- controlling a design tool after it has started, sending it additional commands to execute tasks not anticipated at tool start-up time
- logging execution progress to a design monitor tool to record design history, thus achieving tracing encapsulation (Section 2.8 on page 30).
- linking two tools, e.g. a batch simulator and a graphical design browser, so that the view displayed by the graphical browser always focuses on the currently simulated design objects
- delegating design tasks between tools which allows to structure the design environment into modules

The language processing facility provided by our encapsulation service does not aid in control-integration. Most tool suites and frameworks today, e.g. the Synopsys graphical environment or the Nelsis framework, provide some kind of messaging system based on proprietary message formats. However, standardization is well under way, either as messaging system specific to a particular operating system

(DDE for Microsoft Windows, ToolTalk for SunOS, Softbench for HP), or as an industry standard (CFI ITC, OSF DCE). With the current state of technology, we have to rely on the tools already being able to talk to each other. With regard to control-integration this leads to a situation in which there are tightly integrated tool clusters with well-established inter-tool communication links and more loosely coupled links in between. We also encounter this situation in our design environment. Nelsis has a system of *framework messages* that is used internally to keep the views displayed by the graphical browsers up-to-date with respect to each other and the framework database. Some of the Synopsys tools (debugger, hierarchy browser, waveform display) are integrated with a proprietary messaging system.

As a further option a common look and feel for the design tools and framework tools may be desirable, achieving the lowest level of presentation integration. The most currently achievable here is the use of a Motif-compliant user-interface style for the tools with graphical user interfaces. Both Nelsis' and Synopsys' graphical tools have a Motif-compliant user interface (cf. Figure 30).

7.3.3 Encapsulation tasks

In this section we describe the actual procedures that are used to encapsulate the design tools proper. Two of these procedures (import and export) function as basic services used by all the other procedures. The other procedures serve as wrappers for individual design tools and are more specific to their input and output needs. These procedures are comparable to the tool wrappers used in conventional, file-based tool encapsulation.

Importing VHDL files into the design information management service

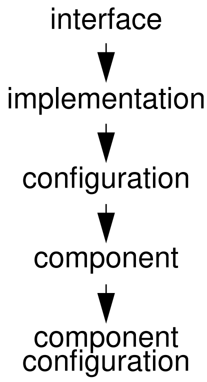
This procedure takes a set of design files in VHDL, extracts the contained design units as text chunks and associates these text chunks with design objects in the design information management database. First, each VHDL file is parsed and a parse tree is constructed for it. The parser is generated from a VHDL language specification as described in Section 6.4.2 on page 112. For each design unit (entity declaration, architecture body, configuration declaration) found, the text of this design unit is replaced by a marker in the corresponding design file as described in Section 5.2.5 on page 78. The remaining file trunk is stored as lexical context for the design objects created for each of these design units.

Then, the parse trees are traversed and the following actions are performed:

- For each entity declaration visited, an interface design object is created. The VHDL text of the entity declaration is written as *interface* stream into this design object.
- For each architecture body visited, an implementation design object is created and inserted into the implementation set of the corresponding interface design object. The VHDL text of the architecture body is written as *implementation* stream into this design object. A default configuration design object is also created and associated with the new implementation. Starting from the architecture body, the parse tree is further searched for component declarations. Each component declaration found is inserted as component into the default configuration.
- Each configuration declaration visited is analysed as described in section Section 6.7.3 on page 127. The result is a new configuration design object for each block configuration found in the configuration declaration. Remember that in the case of configuration design objects the original VHDL is not stored with the object. Rather, all relevant information is resolved into attribute values and relationships in the DIM database.

Export design hierarchies to VHDL files

To export a design hierarchy from the DIM database to VHDL files, the design hierarchy has to be traversed and associated VHDL text has to be emitted. On each level of the hierarchy, the following chain of objects has to be visited:



The component configuration references a design object of either subtype. Interface and implementation design objects are directly associated with their VHDL text chunks. Such a text chunk is inserted into the associated context text and placed into the exported VHDL file. Configurations are unparsed as described in Section 6.7.4 on page 131 and exported. The whole process of exporting a design hierarchy is controlled by a recursive procedure that takes as parameters

- the object references of the root design object *rootId* in the hierarchy and the reference of the currently processed design object *currentId,*
- a regular expression *binaryPattern* for the names of binary streams that have to be exported alongside the actual VHDL design descriptions,
- Tcl code *hierarchyAction* to be executed for every design object in the hierarchy, and
- Tcl code *rootAction* to be executed for the root design object only.

On each hierarchy level, for each interface visited, the interface text is exported. Then an implementation with version status *actual*, or, if none exists, one with version status *working* is visited. For each implementation visited, the interface and implementation texts are exported. If the interface design object contains binary streams whose names match *binaryPattern* these streams are exported as individual files. Then the associated default configuration is visited. For each configuration visited, the interface and implementation texts are exported and the configuration is unparsed and exported.

After all design objects have been exported on a certain level of hierarchy, the recursive export procedure calls itself for each child design object in the current (default) configuration to recursively export all the children in the design hierarchy. Following this recursive descent, on each level of the hierarchy, *hierarchyAction* is executed. If this action results in the creation of new binary files, they are saved as streams in the current implementation design object. If the current design object is the root of the exported hierarchy, *rootAction* is executed after that. Binary files that have not yet been collected by a *hierarchyAction* are collected now and associated with the root object in the exported hierarchy.

The remaining encapsulation tasks operate in the structure imposed by the recursive export procedure and add appropriate code fragments for *hierarchyAction* and *rootAction.* These code fragments may also call the import procedure to import VHDL files resulting from design tool runs in either of these actions.

Analysing VHDL files

To analyse design objects the Synopsys tool *gvan* is invoked in a *hierarchyAction* code fragment. When a syntax error is encountered, *gvan* emits the file name and line number of the error. A text editor is started in which the designer may correct the error. Syntax check and editing is repeated until *gvan* reports no more errors. The corrected VHDL design files are then imported into the DIM database by calling the import procedure. Along with them the **.sim* files generated by *gvan* are saved. These files are needed as input files for the debugger. The import either overrides an existing *working* version or creates a new version of the corrected design object.

Synthesizing structural from behavioural descriptions

Synthesis is conducted by invoking the Synopsys *design_analyser* in a *rootAction* code fragment. Some cooperation is required from the designer as a lot of actions can be performed using the graphical user interface of this tool, resulting in output files not noted by the encapsulation scripts. After successful synthesis, the designer is requested to write out the synthesis result as a binary **.db* file. The export procedure collects these binary files and associates them with the hierarchy root. A separate VHDL generation tool is invoked explicitly by the designer to convert such **.db* files to VHDL text that can be imported into the DIM component. This has the advantage that some iterative synthesis steps can be performed on the intermediate files which are read much faster by the *design_analyser* than are VHDL files. The VHDL generator actually is a script in *design_analyser's* script language, fed into the text-only interface of the tool. The resulting VHDL files are imported into the database as usual.

Debugging a design

Debugging a design is performed by running the Synopsys tool *vhdldbx* in a *rootAction* code fragment. The debugger needs the exported VHDL files only to show currently debugged VHDL text in its text window. The actual design data are read from *.sim files resulting from previous VHDL analysis. Any binary files produced by the debugger are saved along with the hierarchy root but are otherwise left uninterpreted. They can be exported by standard framework means and fed to waveform displays, result comparators or other post processing tools.

	Module		Lines of code		
			Language spec.	Tcl	C++
1	Parse tree implementation				441
	DIM PI on top of Nelsis				971
	Schemas for demand loading			94	
		Parse tree		32	
		DIM		62	
2	VHDL language specification		123		
	VHDL file import			150	
	VHDL file export			127	
3	Execution protocols			211	
		Analyse (calls *gvan* and *vhdldbx*)		36	
		Compile (calls *design_analyser*)		37	
		Derive (calls *dc_shell*)		39	
		Edit (calls text *editor*)		99	

Table 13. *Lines of code in the prototype implementation of the encapsulation service*

7.4 The implementation

The actual implementation of the prototype design environment is structured according to Figure 10 on page 56. It is built around a central Tcl interpreter that provides bindings to the underlying design information manager and to the various modules of the encapsulation service implemented in “C++”. The flow of control and design data between design information manager and encapsulated tools is conducted by Tcl scripts. Table 13 shows the size of the prototype in lines of code. The size of the individual modules is generally small. Modules under (1) are core modules of the encapsulation service and are invariant to supported design description language or encapsulated design tools. Modules under (2) are invariant to design tools but have to be written for each new design description language supported. Only modules under (3) depend on the actual design tools. These modules contain code that is invoked from within the *hierarchyAction* and *rootAction* code blocks of the *export* module. They invoke procedures from the *import* module to import result files in to design information management. To summarize, the amount of code necessary to encapsulate design tools with the aid of our new encapsulation service is similar to what is needed with classical shell wrapper encapsulation. The effort needed to add support for a new design description language is also low.

7.5 Discussion

First, we check to what extent our implementation satisfies the requirements stated in Section 1.6. Requirements **R1** to **R5** are well covered by our conceptual schema. Requirement **R6** is covered by the application of the syntax specification language and associated language processor construction toolkit. It must be emphasized that we do not attempt to improve tool interoperability by writing converters to translate between different design representations. Our main technical goal is to improve interoperability between tools and framework components. This improves transparency between encapsulated and integrated tools in the resulting design system. Finally, by selecting a widely used, leading edge commercial synthesis suite **R7** could in fact be satisfied. We let the analysis of our implementation be guided by the categories of Chapter 2, starting with *granularity.* The following table summarizes the different granularities in our implementation:

The separation of object-based storage and file-based transport of design data between DIM framework service and design tools is a major achievement of our approach. Storage with at least object granularity is an important prerequisite to being able to do design information management at a granularity that is natural to designers and effective to be used when a web of relationships between design objects is to be built and maintained. As mentioned above, this raise in granularity only improves interoperability between framework and tools and does not in itself provide better tool interoperability. In this area we rely on standard design description languages like VHDL. Using such a standard language also ensures tool interchangeability at least as far as design data exchange is concerned.

Unfortunately, our approach contributes little in the area of control integration. In this area tools should provide programmatic access to their inner workings. Although the Synopsys tools do extensive logging of design actions and are busily engaged in inter-process communication, little of this can be used to interfere with a tool once it is started. We have experimented with ways to trace information written into the log files, but with standard i/o buffering one can never be sure if logging information appears synchronously with design processing steps. Also trying to control a running tool by sending it commands during runtime was a frustrating experience. Design analyser, for example, comes with an extensive programming language with a rich set of configuration options, control flow statements, and design commands. Unfortunately, the variant of design analyser with graphical user interface does not provide a channel to feed it commands at run time by another program. To conclude, a minimum of programmable observability and controllability is required in a design tool to achieve any kind of control integration beyond simply starting and stopping of tools. Users find completely new metaphors of employing tools once they are able to attach external programs to them (e.g. [Silva 93]) so even the vendors profit through greater sales. Publishing tools like e.g. FrameMaker (through ONC/RPC) or many MS Windows programs (through DDE and OLE) already provide this kind of openness.

Presentation integration also suffers through this lack of control interoperability. While the Synopsys tools as well as the Nelsis Framework tools are able to update window content amongst themselves with new events arriving, the two tool groups are isolated from each other in this respect.

Mechanism	Granularity
storage	object-based
transport	file-based
processing in the DIM service	object-based
processing in the tools	value-based
browsing	object-based

Table 14. *Granularities applied to different mechanisms*

Finally, process integration could be much finer if only the programming languages offered by the design tools could be used by external programs or wrappers to control the design tools. Currently, only complete sub-functions of the tools as configurable by command line options or initialization scripts can be used as building blocks to construct a design process.

Still, we feel that raising the granularity of data storage in a framework's DIM service is a major step to improving end users' productivity and will help to foster the acceptance of open frameworks with off-the-shelf tools as opposed to single-vendor design systems.

The following table summarises our achievements in the integration categories as introduced in Chapter 2:

Category		Achievement
Inter-operability	framework-tool	Raised granularity
	tool-tool	Not directly supported. Must pre-exist
Inter-changeability		Not directly supported
Dimensions	data	Major asset
	control	Better infrastructure through Tcl extension language. Not exploited in the prototype
	presentation	Not directly supported
	process	Conceptually supported by execution protocols. Not exploited in the prototype due to lack of observability/controllability in the tools

Category		Achievement
Levels	carrier	Original level for file-based encapsulation
	lexical	Lexical contexts
	syntactical	Syntactical contexts
	semantical	Conceptual model for structural information
	method	Not addressed
Granularity		Raised (cf. Table 14)
Observability		Conceptually addressed (cf. “tracing encapsulation”, page 31). Not exploited in the prototype
Controllability		Not addressed

7.6 Related work

To show that this is indeed the case, we look at two other efforts to build integrated design environments. The first example is the Synopsys tool suite [Synopsys] itself, with its extensive range of tools for design synthesis and simulation. Several standard design description languages interface the tool suite to the outside world, amongst them VHDL, Verilog, and EDIF. We will only look at VHDL here. In the Synopsys tools suite there are two major tool clusters, namely the design compiler for synthesis and the simulators and graphical browsing environment with symbol and schematic editors for browsing and simulation. Data exchange between the two clusters is via two alternative paths:

- Opaque **.db* files created by the design compiler are used to transport graphical information such as generated schematics from the design compiler to the browsing and simulation subsystem. These files have to be converted by a tool *db2sge* into the native format of the browsing subsystem.
- Behaviour and structural design information are exchanged between the two clusters via VHDL files. While the design compiler directly reads VHDL files, simulator and debugger rely on intermediate **.sim* files produced by an analysis tool.

There is no control integration between the two tool clusters. Within the browsing and simulation environment, a lot of control messages are exchanged between schematic editor, waveform displayer, and debugger to maintain the illusion of different views on single design objects. The message types and format is unpublished and so can not be exploited for framework integration.

With respect to design information management, the design compiler largely ignores configurations in its VHDL input but applies simplified, default bindings by name. It can, however, generate proper VHDL configuration statements as output which can be read by the debugger and simulator. By supporting configurations already in the DIM component and resolving them on export, we can make these differences in configuration support in different tools largely transparent to the designer.

Another project which is interesting to compare to our approach is the effort to encapsulate some Cadence tools into the JESSI Common Framework (JCF, [JCF 94b]). The encapsulation relies entirely on file-based storage and transport between the JCF and encapsulated Cadence tools. While JCF only provides file-based granularity, Cadence tools are integrated into the Cadence Opus framework at object- to value-granularities. The encapsulation relies on the standard JCF approach (outlined in Section 2.8) to export a set of predefined files from a framework-managed cell's pool of design descriptions, to invoke some shell wrapper and have it pass control to the design tool, and finally to collect the results. Once control has been passed to the design tool, no communication happens between the tool and the framework until the tool is finished. All design data management on the Cadence side is volatile and effects only the current tool run. The resulting prototype design environment proved unusable for all practical purposes.