Design Tool Encapsulation - All Problems Solved?[*]

O. Schettler
Olav.Schettler@GMD.de

GMD, Sankt Augustin, Germany

Abstract

Although a prime goal of CAD frameworks is to facilitate cost effective, efficient, and seamless incorporation of tools into design systems little support is given to tool integrators. We present a new methodology called execution protocols that allows to abstract tool integration from a particular framework or tool. The actual design step is performed by an execution protocol engine that is guided by language and communication processors generated from an abstract specification of the relevant information content of data and start-up files. The execution protocol methodology may be regarded as a natural evolutionary step from today's wrapper encapsulation.

1. Introduction

One goal of CAD frameworks is to "facilitate cost effective, efficient, seamless incorporation of tools into design systems". Objectives to reach this goal are interoperability, interchangeability, abstraction, integration, and encapsulation of tools (CAD Framework Initiative, [UGO092]). An underlying assumption in CFI's work is that a *single*, standard framework architecture can be defined. This goal may in fact not be reachable due to the dynamics in the ECAD domain and in the workstation market:

- *Application domain*
  - Advances in design methodology and a temptation to work on higher levels of abstraction (behavioural, architectural) require a constant evolution of design languages and their associated language processors.
  - The advent of concurrent engineering has broadened the scope of design systems from pure ASIC or PCB design to e.g. mechatronic systems design with associated design language 'standards'.
  - New and broadened application domains require new graphical presentation techniques (schematics, state charts, time vs. frequency domain) and new forms of tool interaction.
- *Workstation technology:*
  - *Data integration:* Advances in object-oriented database technology (e.g. memory-mapped architectures) can efficiently handle the main memory data structures of design tools.
  - *Control integration:* Recent operating system enhancements provide direct support for broadcast messaging (SUN's tool-talk).
  - *Presentation integration:* Embeddable extension languages like *Tcl* and *Scheme*, powerful UI toolkits like *Tcl/Tk* [Ousterhout91], versatile graph layout systems like *edge*, *graphedit*, or *dot* [Koutsofios93] and standardization efforts like *COSE* make dedicated framework developments in this area obsolete.
  - *Process integration:* Process engines with open interfaces emerge (*CCS* [Milner89], *Marvel* [Kaiser88]) and are likely to be more versatile than dedicated framework developments.

Framework supported, fine-grained design data handling does not seem feasible and, in fact, desirable for the following two reasons:

- Whereas design tools are centred around a specific and detailed understanding of the design objects they manipulate a framework needs only a coarse-grained perspective to provide execution contexts for its integrated tools. 'Coarse-grained' is not to be confused with 'file-based'.
- Design tools are written to an up-to-date model of the design objects they handle. In fact, all too often a particular design language dialect is exclusively defined by the sole tool that works on it. A framework is always the second to come and will rarely have exactly the same model used by the tools it serves. Keeping the model up-to-date is expensive, and hardly worth the effort. Rather than to define an omnipotent framework with general purpose but either low level or inflexible interfaces we advertise an approach based on software module generation that can react more flexibly to new requirements.

In this paper we present a new methodology called *execution protocols* that allows to abstract tool integration from a particular framework or tool. The actual design step is performed by an *execution protocol engine*.

[*]. This research was supported in part by the commission of EC under ESPRIT contract 7364

The engine may be regarded as an advanced form of wrapper that has access (up to some level of granularity) to the contents of design files. Guided by information passed from the framework and additional information extracted from affected design files the engine may request additional design objects from the framework. For this, it must have a certain understanding of the contents of the design files it handles to determine the dependencies between described design entities. It is, however, not necessary for the engine to process design files down to the finest detail. Only the actual design tools need this. The engine then prepares start-up files, sets up the execution context for a design tool, runs the tool and finally collects and processes the results to pass back to the framework.

In section 2 we introduce execution protocols. Section 3 introduces design file processing in frameworks. Section 4 explains how to generate language processors used by the engine. We describe the execution protocol engine in section 5, and give our conclusions and ideas for further refinements in the last section.

2. Execution Protocols

Integration methodology in current framework approaches (e.g. [Kathöfer92]) assumes the following, fixed execution sequence to perform a design step:

1. The framework checks out a cell hierarchy into the file system.
2. The framework invokes a wrapper shell-script, passing it environment variables and the name of a design object as command line parameters.
3. The wrapper preprocesses and copies the design files to appropriate places, sets additional environment variables, assembles a command line and invokes a design tool with it.
4. The design tool performs the actual design step, resulting in success or failure.
5. The wrapper checks this result to decide on a number of post-processing alternatives, again processes and copies files and finally informs the framework of success or failure of this particular design step.
6. The framework in turn checks in the resulting design files and decides on their status depending on success or failure of the design step.

This sequence shows that most of the actions necessary to start a design tool are performed in the wrapper. The art of tool integration is reduced to tedious shell programming like in the early days of UNIX. But there is another, more important drawback of this simplistic approach. *All control is on the side of the framework.* Once the wrapper is started, no more design objects may be requested from the tool side. Also, *all* result files have to be determined in advance, making it necessary to split generic design tools with complex input/output relations into myriads of 'activities', one for each combination of input/output files.

Our approach replaces the simple wrappers with an engine that realizes an abstract machine for execution protocols. An execution protocol is comparable to a handshake protocol used in networking to define the state transitions of communicating agents. The next section introduces the agents in a simple design system.

2.1 A Simple Design System

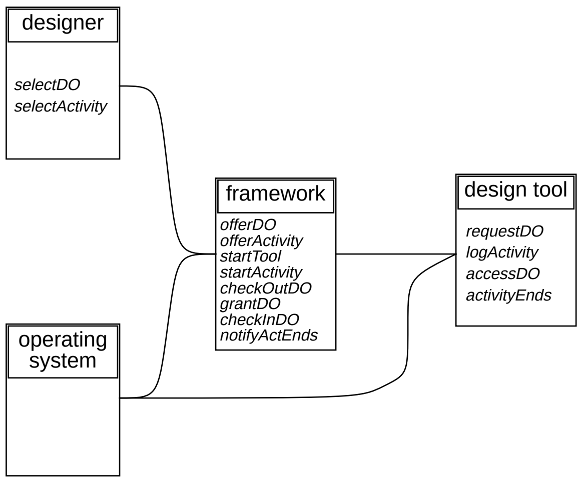


Figure 1. Model of a simple design system

Figure 1 gives an overview of the communicating objects or 'agents' that send and receive messages in a simple execution protocol. Each of the agents implements a number of methods:

1. *Designer*

selectDO	use any of the means offered by the framework to select a design object
selectActivity	select an activity to apply to the design object

2. *Framework*

- *User Interface*

offerDO	offer a design object for manipulation
offerActivity	offer an activity depending on the design object type
notifyActEnds	present the result of the activity

- *Design Data Handler*

checkOutDO	check out a design object (flat/hierarchical) into the file system
grantDO	notify the availability of a design object
checkInDO	check in a design object from the file system

- *Tool Server*

startTool	start a tool or connect to running tool
startActivity	start an activity within a tool

3. *Operating System*

- *Process Manager*

  /* start/stop/signal tools */

- *File System*

  /* store files hierarchically */

4. *Design Tool*

- *Activity*

requestDO	ask for a designated design object
logActivity	write log about the activity: execution time, execution environment, affected design objects
accessDO	read a design object extracting its information contents
activityEnds	signal success/failure of the activity

2.2 A Sample Execution Protocol

	agent	recipient	message	parameters
1.	f/w[a]	designer	offerDO	set<DO[b]>
2.	designer	f/w	selectDO	DO
3.	f/w	designer	offerActivity	set<activity>
4.	designer	f/w	selectActivity	activity
5.	f/w	o/s[c]	startTool	tool-path
6.	f/w	tool	startActivity	activity options DO
7.	tool	f/w	requestDO	DO placement flat/hierarchy
8.	f/w	o/s	checkOutDO	placement
9.	f/w	tool	grantDO	DO placement
10.	tool	f/w	logActivity	DO activity
11.	tool	o/s	open/read/ write/close	DO-path
12.	tool	f/w	activityEnds	DO activity
13.	f/w	o/s	checkInDO	placement
14.	f/w	user	notify ActivityEnds	DO activity

Table 1. A sample execution protocol

a. framework

b. design object

c. operating system

Table 1 depicts a sample execution protocol. The framework offers the designer a set of design objects through its user interface. After selecting, one out of a set of activities (methods of the selected object’s type) becomes available. The framework finds the design tool associated with the selected activity, starts it if necessary and sends it the request for the activity, passing the identifier of the selected design object as parameter. The tool requests the denoted design object from the framework, passing a desired placement in the file system. When the checked-out design object is granted and placed in the file system the tool starts the requested activity, requesting additional design objects when needed. On completion the framework is notified of success/failure. The framework checks in the results and notifies the designer of activity completion.

3. Processing Design Files

One of the foremost features of a framework-based, integrated design environment is to offer design management functions to the designer. Following the assumption that commercial design tools interface to design descriptions stored in files, in this section we introduce the necessary steps to extract structural information and design object representations from design files on import and to reconstruct valid design files on export from the framework. Using this approach it is possible to exploit the management features of a framework while at the same time being able to use commercial, file based tools.

3.1 Exporting a Compound Design

At certain points in a design process it is necessary to create a textual image of the design managed by the framework. The most obvious such point is when an encapsulated design tool is run that is not aware of the framework but rather expects to find a certain arrangement of design files in the file system. Other occasions might be the export of (partial) designs to another design system or the printout of a text version of the design. What is basically required is some kind of traversal through the composition hierarchy managed by the framework, starting at a selected design object, which might be an interface, a configuration, or an implementation of a module. Depending on the design language used or on the language processing capabilities of the design tool invoked, the traversal has to be bottom-up to emit sub-modules before they are used as components. As the composition hierarchy is traversed, design representations are emitted into design files, enclosed with any global text that was saved on import. Depending on design tool requirements, single design objects are placed in each emitted file, interfaces are combined with their associated implementations and configurations, or even a single large file is emitted.

The existence of configurations in a design hierarchy complicates matters. If a design description language is used that supports configurations (like VHDL), the configurations managed by the framework can be translated into appropriate configuration declarations in the language and emitted along with the selected interfaces and implementations. When, however, the design language does not have a notion of “configuration” or the one supported by the language grossly deviates from the notion used by the framework, configurations have to be resolved and emitted as static bindings between compound modules and their components.

3.2 Importing Designs Files

To import design files after a successful tool run, structural information has to be separated from design representation. The structural information is managed by the framework as a graph of typed objects and their relationships according to a design management schema. We assume that design representation is attached to nodes in this graph but is otherwise left uninterpreted. The graph can be inspected and queried with the browsers offered by the framework.

Exactly which structural information can be extracted from a set of design files, how it is organized and how design representation is attached to it, is of course highly dependent on the design description language used. In fact, the parsing and analysis of design files is very similar to the initial parsing phase of a compiler for that particular design description language. There are, however, also significant differences:

- The lexical and syntax analysis in a compiler tries to digest every single detail of a design description to be able to derive an internal representation that matches the information content of the design description as thoroughly as possible. For the purpose of design tool encapsulation, only a small part of design files need to be analysed down to single lexems. Large parts can simply be skipped and regarded as design representation, uninterpreted as far as design management is concerned.
- A large part of the lexical and syntax analysis phase of a compiler is concerned with error detection and recovery. As this tedious work is already performed by design tools, we can rely on a design description being syntactically correct.
- Another big effort in the initial phase of a compiler is the construction and maintenance of a symbol table for semantic analysis. We are, however, not interested in things like data or control flow and can therefore greatly reduce the effort of symbol handling. We also store extracted information directly in a framework with versatile querying facilities, so manually constructing a symbol table is not required.

On the other hand, there are also requirements in the realm of tool encapsulation that are not an issue with ordinary compiler technology:

- All text that is considered design representation does not need to be interpreted in any way, but it must be saved completely for later perusal. Once an ordinary compiler has built an internal representation of the input read it can safely forget about the exact textual representation, maybe with the exception of line numbers for error messages.
- Global text that neither belongs to structural information nor is associated with single design representation chunks needs to be preserved to be able to reconstruct complete design files later. In a compiler this text is sometimes already resolved by a preprocessor and not even seen by the actual compiler, or it serves as context information and is also translated to objects in an internal representation.

3.3 Associating Binary Objects with Structural Information

The import and export mechanisms described above assume that design files have a well-defined structure and contain a printable description of the design. Quite often, however, design tools produce and expect some kind of opaque, intermediate design description. Examples for this kind of files are results from VHDL analysers, simulation results, or schematics in undocumented, proprietary formats. When no specification of the lexical and syntactical structure of such opaque descriptions is available to the tool integrator the files containing these descriptions can only be manipulated as a whole. If such files can be associated with a specific design object in a composition hierarchy, their contents can be attached to this object and will be imported and exported together with it. If it cannot be associated with a specific design object it has to be attached to the root object of the composition hierarchy and imported and exported whenever a sub-module of the root object is imported or exported.

4. A Specification Format for Language Processors

The execution protocol engine needs to read design files to extract dependencies between design objects contained therein. Traditionally this is accomplished by any of the following alternatives, ranging from little coding effort with only crude recognition capabilities and high probability of failure due to small syntactic variations in the processed design files to prohibitively high coding effort with detailed recognition capabilities but with high dependence on fine-grained detail of the processed syntax.

1. combination of UNIX tools like *sed/awk*, or more in vogue, *perl*
2. *lex* generated lexical scanner
3. *yacc* generated parser
4. *yacc* generated parser enhanced with semantic actions to resolve references

We have defined a language to facilitate the specification of design language processors used by the execution protocol engine. The language combines syntax specification features of scanner and parser generators and is geared towards the construction of an abstract syntax tree that is easily queried and traversed from within execution protocols. The design is tailored to meet the following requirements.

R1 Provide a single, coherent notation for both lexical and syntactical features.

R2 Allow to process a design file at varying granularity. Certain regions may be processed down to single lexems whereas other regions may simply be skipped.

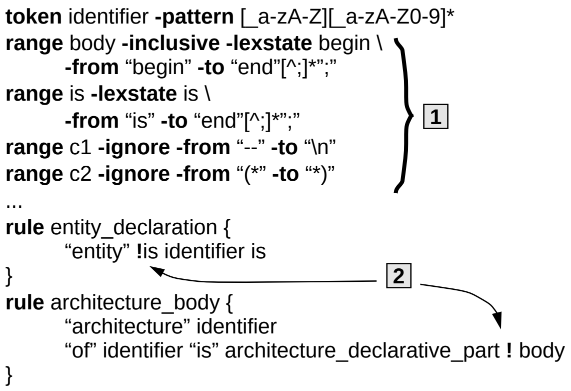
R3 Register information loss. Design objects which only partly read should be recognizable as such.

R4 Manipulate multiple files with single or multiple syntaxes. Often tool start-up files have to be consulted to derive the expected physical location of design objects denoted in a design file.

R5 Allow arbitrary semantic processing (queries and traversal) on an explicit parse tree. Since much information in the input will simply be skipped, the ease in formulating semantic actions by far outweighs the cost of maintaining an explicit parse tree.

R6 Allow to easily split/recombine design files into/from a hierarchy of chunks. Both the framework and the design tool may expect different combinations of design objects in a single design file.

We assume that in most cases a grammar for a design language is contained in its definition and will be taken as the basis for the specification of processors for this language. The lexical properties and syntax of a design language are invariant to the task of tool encapsulation and can be fed into conventional scanner and parser generators to generate a scanner and parser for the language. The extraction of design objects and their dependencies, however, does depend on the specific encapsulation task to be accomplished. Whereas the lexical and syntax processing is performed by a compiled module, the extraction steps are performed in execution protocols and are to be interpreted by the execution protocol engine.



```
token identifier -pattern [_a-zA-Z][_a-zA-Z0-9]*
range body -inclusive -lexstate begin \
        -from “begin” -to “end”[^;]*”;”
range is -lexstate is \
        -from “is” -to “end”[^;]*”;”
range c1 -ignore -from “--” -to “\n”
range c2 -ignore -from “(*” -to “*)”
...
rule entity_declaration {
        “entity” !is identifier is
}
rule architecture_body {
        “architecture” identifier
        “of” identifier “is” architecture_declarative_part ! body
}
```

Figure 2. Excerpt from a language specification for VHDL

A syntax specification in our specification language is structured into lexical declarations and syntax rules. Figure 2 shows an excerpt from such a language specification. This example demonstrates the following features of the specification language (numbers refer to figure 2).

- Lexical and syntactical features are specified in a single file (1). While this is more a convenience feature for fixed tokens like “entity” or “;”, the possibility to define ranges and scanner states is a key feature of the language. Range definitions follow the keyword *range* which may be qualified by properties *-inclusive*, *-ignored*, or *-nested*. A range is defined by giving its name, optionally a scanner state in which the range should be recognized, and two regular expressions for its start and end. The lexical declarations section also allows to declare comment formats, which are in fact just special ranges that are invisible to syntax recognition.
- Syntax rules may reference other rules, primitive tokens, or ranges on their right hand side, using an EBNF-like notation.
- Lexical states may be switched anywhere on the right-hand-side of a rule (2). A shorthand notation is available to allow prefixing a range reference with a “!” to switch to its associated lexical state. When a rule is completely recognized, lexical state is automatically switched back to the initial state. Care has to be taken that state switching does not interfere with the generated parser’s look-ahead symbol.

The example does not show language constructs to maintain a stack of input files for processing multiple design files.

We have implemented a generator that translates such a syntax specification into specifications for yacc/lex and a module that implements the execution protocol statements to manipulate an abstract syntax tree of the design description. Design files are processed in two phases. The first phase constructs an in-core representation of the abstract syntax tree. The tree is constructed from generic nodes that are associated with the originating symbol as node type, the start and end offsets of its associated text region, and references to children rsp. list elements.

The second phase consists of execution protocol statements to query and traverse this syntax tree. As the tree is built during the language parsing by automatically generated code, it contains much detail that is irrelevant to an execution protocol run. The execution protocol statements therefore are defined so that they allow to only look at “interesting” nodes. Two statements are defined:

1. node **all -type** node-type **-var** var code-block
2. node **one -type** node-type

Statements (1) execute *code-block* for every node of type *node-type* in the tree rooted at *node*. The node currently looked at is available in the variable named *var* within *code-block*. *Code-block* may contain *break* and *continue* statements to break out of the traversal completely rsp. to continue with a sibling of the current node. Statement (2) traverses the tree rooted at *node* and breaks at the first node of type *node-type* or fails if no such node exists.

5. The Execution Protocol Engine

As a descendent of shell wrappers the execution protocol engine must be able to perform the following tasks:

- Process parameters passed by the framework to determine affected design objects
- Establish an initial execution context, e.g. by creating a temporary directory and creating invariant tool start-up scripts
- Request affected design objects from the framework
- Traverse the composition hierarchy of selected design objects into the file system, requesting additional design objects on the way to construct valid design files
- Finalize the execution context by creating/modifying start-up scripts and setting up the environment
- Assemble a command line and invoke the design tool
- Establish a message connection to an already running tool; invoke an activity in it
- Collect created/modified design files
- Extract additional information from the result files to get a more detailed notion of design step status (success ... failure)
- Check-in result files into the framework and update design management information related to them
- Inform the framework about the design step status
- Close down running design tools
- Clean up execution context: remove temporary files/directories, free other system resources like displays, plotters, etc.

As can be seen from this list, a great deal of flexibility is required by the execution protocol engine. We have chosen to base its design on an embeddable language kernel [Ousterhout91] that already provides constructs for control statements, modules, procedures, data types, and variables. This kernel is extended with demand-loadable modules that realize dedicated functionality. The core of each module is generated from a data schema and creates and maintains objects. Some object methods invoke communication primitives to exchange messages with the framework or design tool. Other methods are generated from language processor specifications to allow design file manipulations.

Execution protocols are implemented on this engine by writing scripts comprising of kernel language constructs and commands that invoke functionality residing in a demand-loadable module. In standard cases where similar tools or execution protocols have already been realized, no compilation is necessary to completely integrate a design tool by implementing a new execution protocol. Only in cases where either new communication primitives or a new design language dialect have to be supported a new module has to be generated from a data schema and a language specification.

6. Conclusions

Although powerful frameworks have emerged in the ECAD arena, even loosely coupled tool encapsulation is still tedious, resulting in inflexible and hardly comprehensible solutions. The flexibility lost is all too often an excuse to neglect the merits a framework has to offer to the designer but rather stick to traditional means of tool invocation and manual version handling and configuration management.

We have presented a new methodology that simplifies the process of design tool integration by (1) reusable execution protocols, (2) abstract specification of relevant information in design files, and (3) easy-to-use generation of demand-loadable modules which realize specialized language processing or communication primitives.

This methodology was evaluated by creating language processors for VHDL design files, symbol libraries, and start-up files and, using these, incorporating selected tools from the Synopsys synthesis and simulation tool suite into a framework providing check-in/check-out of design files and general design management facilities. Generated language processors were mainly used to derive “system models” i.e. dependencies between the various files manipulated by the design tools. The resulting system models were more detailed than the ones produced with the *simdepends* tool provided by Synopsys because our generated language processors can consult more relevant information than what seems to be used by *simdepends*.

References

[Kaiser88]
Kaiser, G.E.; Feiler, P.H.; Popovich, S.S., *“Intelligent Assistance for Software Development and Maintenance”*,
IEEE Software, pp.40-49, May 1988.

[Kathöfer92]
Th. Kathöfer, J. Miller, *“The JESSI-COMMON-FRAMEWORK Project - Subproject Development -”*, in: T. Rhyne ed., Electronic Design Automation Frameworks, Elsevier Science Publishers B.V. (North-Holland), 1992

[Koutsofios93]
Eleftherios Koutsofios, Stephen C. North, *“Drawing graphs with dot”*, dot User’s Manual, AT&T Bell Laboratories, Murray Hill, NJ, June 22, 1993
URL=ftp://research.att.com/dist/drawdag/dotdoc.ps.Z

[Milner89]
Milner, R., *“Communication and Concurrency”*,
Prentice Hall, 1989

[Ousterhout91]
John Ousterhout, *“An X11 Toolkit Based on the Tcl Language”*, *Proc. USENIX Winter Conference*, January 1991
URL=ftp://sprite.berkeley.edu/tcl/tkUsenix91.ps

[UGO092]
CAD Framework Initiative, Architecture TC, *“CAD Framework - Users, Goals, and Objectives”*, Version 0.92, 1990