2. Design tool integration

2.1 Overview

In this section we will review classification schemes used to rate the degree of tool integration achieved in a particular design environment. Unfortunately, the diversity of possible viewpoints taken by researchers and the importance of a good rating for the marketing of an integrated design environment resulted in a muddle of different classification schemes. Classification schemes can be discussed from two points of view, the environment user's and the environment builder's:

> *The environment user is concerned with perceived integration at the environment's interface. [...] The environment builder, who assembles and integrates tools, is concerned with the feasibility and effort needed to achieve this perceived integration.*
> *[Thomas 92], p. 32*

Bad integration[*] from an environment user's point of view results in higher costs during the actual design using the environment. Bad integration from an environment builder's point of view results in high costs for environment construction. This distinction is important because it enables an environment builder to make a set of tools appear well integrated to a user by providing some "glue" between the tools, even if it is not well integrated from his point of view. Thus, although the tools were not built to work in a particular design environment, this glue between tools and between tools and framework services can help to make a design environment better integrated at least from the user's point of view. Of course, it would be more elegant to have all the components of a design environment well integrated in the first place. However, this requires mature, flexible, and standard interfaces to all components in an integrated design environment. Defining such interfaces is a tedious process and due to changing requirements and technology will not be achieved in the near future, if ever.

The goal of using an *integrated* design environment as opposed to a set of independent design tools is enhanced design productivity. In Sections 2.2 and 2.3 we introduce two notions which are key to enhanced productivity in an integrated design environment, namely tool *inter-operability* and tool *inter-changeability.* Tools interoperate in a multi-user design environment when they can freely communicate and share *at least* control information and data with other tools and framework services.

[*] according to any of the classification schemes introduced in this chapter

A tool should be easily inter-changeable with another tool, perhaps from another vendor, to supply the same function. This way a design environment can be upgraded to offer state-of-the-art tools for a design task at hand.

The well-established distinction between the orthogonal dimensions and levels of tool integration are discussed in Sections 2.4 and 2.5. Two subsequent sections are devoted to the more controversial aspects, granularity of control and observation, and the black-to-white scale of integration mechanisms. Based on these taxonomies, in Section 2.8 we examine the level of tool observability and controllability that can be achieved by different approaches to tool encapsulation. A synopsis of related work and our conclusions lead to three principles that will serve as the basis of the tool encapsulation methodology to be introduced in the subsequent chapters.

2.2 Interoperability

There are four mechanisms of design data handling allowing to supplement or completely replace conventional operating system services by framework technology:

- *Storage* is concerned with persistent archiving of design data and efficient retrieval of this data.
- *Transport* is concerned with the movement of design data between environment components, which may be design tools or framework components.
- *Processing* is concerned with the transformation of design data, either as part of a design step or in preparation of a design step.
- *Browsing* is a technology that ...

> *"... allows engineers to explore the high-level structure of their design. [...] Browsers are important in such an environment because they reveal unobserved, or forgotten structure, enhance design re-usability by making previously designed components easy to find, and assist in communication and shared understanding between different members of a design team."*
> *[Gedye 88]*

Early attempts to apply framework technology focused on enhancing *tool-to-framework* inter-operability (e.g., the Oct System [Harrison86]). A common approach was to build a central database with a global schema that allows to store and access all aspects of a design. Design data was then accessed by tools through a programming interface, processed in core, and then written back to the database. Transport is not an issue for database-oriented systems because design data is accessed in

small portions directly via a procedural interface. Browsing of design data is ideally supported because all aspects of the design are randomly accessible.

Much work has been invested in trying to use relational database systems for design data storage and access. As the performance of relational database proved inadequate for the navigation-oriented access patterns of most design tools, special-purpose design data handlers emerged. Most of the major CAD environment vendors today base their integrated environment products on such special purpose design data handlers. Their success probably stems largely from the fact that the integrated tools were directly developed for the particular design data handler in use and so can optimally exploit its features. Lack of standards, unfortunately, has up to now prevented a wider use of (object-oriented) database-oriented systems despite their obvious advantages.

Tool vendors, and those users that depend on the openness of their design environments, still prefer file based design data storage, transport and processing. The emergence of standard design description languages like VHDL has removed the importance of framework supported data handling to achieve good *tool-to-tool* interoperability. Tools can be written in a framework-independent manner, relying solely on fairly stable standard languages as interface specification. Storage and transport of files is well supported by operating system services, so that hardly any support is needed by enhanced framework services. Manually creating design file processors is tedious but can be obviated by the application of efficient parsing techniques provided by widely available compiler construction toolkits.[*]

Unfortunately, with file based design data storage and transport, *browsing* becomes a major hassle. Browsable relationships between design objects are now obscured within files and have to be extracted. The extracted structural information then has to be kept up-to-date with respect to the design representation stored in files. A common conclusion by framework vendors is to condemn file-based design tools

[*] There is an interesting divergence in the use of languages as opposed to programming interfaces in the UNIX and MS-DOS worlds. Under UNIX, versatile language processing tools have always been available. In addition, the UNIX tool suite is specially oriented towards text processing. The use of text files therefore is common-place under UNIX. By contrast, under MS-DOS language and text processing tools are not available. Also, the fact that all programs share the same address space makes it easy to use public programming interfaces across program boarders. The net result of this fundamental difference in operating system philosophy is that under MS-DOS and its offspring MS-Windows, programming interfaces are versatile and widely accepted by users as well as tool vendors. Under UNIX, text files are the interface metaphor of choice. Only recently, with industry standards like ToolTalk, there is a chance that this situation may change.

altogether and to put off environment builders until database technology is more mature. We want to show in this thesis that with some care even with today's technology effective browsing of design data can be achieved. The important idea is to clearly *distinguish storage from transport mechanisms.* While design tools rely on design data being transported as files, they need not be stored this way. We will discuss this in more detail in Section 2.6 where we take a closer look at granularities.

2.3 Interchangeability

As CFI states, ...

> *"... design System builders tend to judge a tool strictly on its merits as a tool, not on its compatibility with a proprietary framework. They want the freedom to select the best tool for their needs. In current systems, the process of selecting a tool is strongly influenced by the proprietary framework in which the tool operates. In many situations, the tool of choice cannot be easily coupled to the Design System currently in use. Users want to divorce decisions about tools from decisions about frameworks. This will result in a more seamless Design System for the user."*
> *[CFI-UGO 90], O2.2, p. 17*

There are two aspects of design tool interchangeability: *Open interfaces* and *design tool abstraction.* The common, standard framework backplane that CFI suggested in 1990 to solve the lack of open interfaces has neither been specified nor realized until now. Rather than waiting for its coming, users and independent tool vendors have moved to standard design description languages to solve their tool integration problems.

> *"Tool abstraction is the process of reducing the specific details of successfully invoking and executing a tool to a common set of parameters understood by the framework and the user. Tool abstraction aids the designer both in selecting the proper tool for a particular function and in correctly invoking the tool without requiring intimate knowledge of a tool's specific invocation syntax or execution environment."*
> *[CFI-UGO 90], O2.3, p. 18*

Furthermore, tool abstraction can be used to register a design tool with a framework so that different framework components and other tools can effectively coordinate with the new tool ([ECMA 91], p. 11). The success of CFI's proposed standard tool abstraction mechanism is documented by the fact that commercial (e.g., ViewLogic) as well as research frameworks (e.g., Nelsis) have adopted CFI's Tool

Encapsulation Specification format as the method of choice for registering new tools in the system.

2.4 Integration dimensions

It is commonly agreed that there are orthogonal dimensions to integration. In accordance with Wasserman we identify five dimensions [Wasserman 90]:

- *Framework integration* or its weaker form, *platform integration,* is concerned with the effective use of a platform's services. Its goal is to provide the basic elements on which agreement policies and usage conventions are built. The framework is considered an extension of the operating system platform.
- *Data integration* is concerned with the use of data by tools and framework services. Its goal is to ensure that all the information in the environment is managed as a consistent whole, regardless of how parts of it are operated on and transformed.
- *Control integration* is concerned with communication and inter-operation among tools and between tools and framework services. Its goal is to allow the flexible combination of an environment's functions, according to project preferences and driven by the underlying processes the environment supports.
- *Presentation integration* is concerned with user interaction. Its goal is to improve the efficiency and effectiveness of the users' interaction with the environment in fulfilling his design tasks by avoiding distraction through incompatible visual presentation paradigms.
- *Process integration* is concerned with the role of tasks and tools in the design process. Its goal is to ensure that tools interact efficiently in support of a defined process. A process is *"a specific combination of tools and/or other processes that performs a design function"* [Fiduk 90]

If we interpret the integration dimensions as spatial axes spanning an n-dimensional cube, we can associate a complete design environment, clusters of well integrated services, or single tools with a point in this space, depending on our focus (Figure 4). The coordinates of such a dot give a measurement of the level of integration of a given item in each of the integration dimensions. As no quantitative measure exists for the level of integration, such a diagram should only be regarded as a rough statement of the level of integration of two or more items relative to each other.

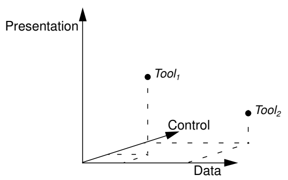

***Figure 4.** Dimensions of integration. Each axis represents a dimension. The coordinates of a point in the spanned, n-dimensional "integration space" can be used as measurement for the level of integration in each of the dimensions for the item represented by the dot. For example, $tool_1$ is hardly integrated with respect to control and data, but well integrated with respect to presentation. The integration of $tool_2$ focuses on data and control integration and neglects presentation integration.*

This diagram leads to the conclusion that the integration dimensions are all equally important in constructing a design environment. Emphasizing either of the dimensions would therefore be more a matter of personal taste than dictated by added value to environment users. When looking more closely at this added value, however, it becomes clear that this is a misconception. Although the sum of integration levels for tools 1 and 2 in Figure 4 may be the same, due to its lack of data and control integration the integration of $tool_1$ may be of less value to a designer's productivity than the integration of $tool_2$. In fact, the integration dimensions can be arranged in a sequence according to their importance in achieving an overall high level of integration, ranging from essential to merely desirable:

1. *Framework integration* provides the technical foundation for all other integration dimensions. Thus, without framework integration or its weaker form, platform integration, there is not much point in trying integration in any of the other dimensions. It is so basic that many authors do not regard framework integration as one of the integration dimensions at all but rather as one of the invariants on which all other integration dimensions are based.
2. *Data integration* is the next important integration dimension. Without data integration, data as the output of one design tool has to be converted to the format expected by another tool. Before the advent of standard design description languages like VHDL or EDIF, writing the necessary converters

was the most expensive effort during the construction of an integrated design environment from off-the-shelf design tools and framework components. This effort was often so trying that achieving good data integration was considered the only important problem in tool integration (assuming a common platform), whereas the other integration dimensions were completely neglected.* It was hoped that by using a database management system (DBMS) for design data storage, all data integration problems could be easily solved, but the use of a DBMS merely transferred the need to lexically and syntactically process design descriptions. The need for agreed semantics between the design tools still remains and the design of a new common conceptual schema for each new DBMS-based integration project binds much of the resources available in such a project. As already mentioned above, a major drawback of using a database management system is that it is hard to write stand-alone tools. This is the reason why DBMS-based data integration is mostly used in proprietary systems and is hardly used by independent tool vendors.
3. *Control integration* can be subdivided into *provision* and *use*. Without control integration, automation is not possible because every single activity has to be triggered explicitly by the user. With control integration, however, each component in the environment can be designed to either *notify* other components of relevant events or to *control* their operations. Hence, efficient control integration allows to construct new complex services or tools from existing ones, fostering modular environment construction. For tools to share functionality, they must be able to communicate the operations to be performed. As operations require data, the tools must also communicate data or data references. Control integration therefore relies on the existence of data integration.
4. *Presentation integration* not only involves compliance of the user interfaces of tools with a common appearance and behaviour, it also involves that tools update their user interfaces to always give a consistent view on the current design state. To accomplish this, tools have to be able to receive notifications from framework services, thus relying on control integration.

*. Compare the results of the German DASSY project (Data interchange and interfaces for open, integrated design systems) as documented, for example, in [Hunzelmann 92], a joint effort of German universities and industrial research institutes to define a common tool interface. Despite the project's claim to cover all aspects of tool integration only data integration was considered in depth.

5. *Process integration* relies on both data and control integration. Sequencing of design tasks would not make much sense if tools were unable to exchange design data. Automatically running processes require that tools are able to react to events that reflect their preconditions in a process. Tools should be able to generate events that help to satisfy other tools' preconditions. This requires control integration. Furthermore, presentation integration is important when a running process is to be monitored by a browser tool.

As long as the only interface to state-of-the-art design tools is via design description files, only proprietary systems will be able to exploit the full range of integration dimensions. A tool that internally processes design objects will not be able to communicate its need for additional objects to a data management service that only recognizes files, not the objects hidden in these files. Design browsers will not be able to display relationships between design objects described in the same design file. Design constraints attached to certain design objects will not be able to trigger necessary events when only the modification date of a file changes as the result of an update operation but nothing is known of the changes that were made to the file's content.

All of these deficiencies are due to the fact that design information is hidden away in design files. Although we can expect that we will have to deal with design files for quite a time, much higher levels of presentation and even process integration can be accomplished if the framework components recognized the contents of the design files they handle.

2.5 Integration levels

In the last section we frequently mentioned the level of integration in each of the dimensions. Although there is no quantitative measure for these levels, the introduction of a coarse classification scheme can help to compare one specific integration with others. This classification scheme was introduced by Brown and McDermid [Brown 92], who have also defined several integration dimensions that somewhat deviate from the more common terms introduced above. They use

- *tool integration* to refer to a combination of our data and control integration,
- *interface integration* to refer to our presentation integration, and
- *process integration* in accordance with our terminology.

Their dimensions *team integration* and *management integration* have no resemblance in our system. Brown and McDermid apply integration levels only to data and control integration: they remark that obstacles in these dimensions must be removed first before the presentation and process dimensions can be explored in more detail. We extend their notions to presentation integration.

1. *Carrier level integration* is concerned with the basic physical prerequisites of integration. For data integration, this e.g. would mean to agree on using physical files without providing any common module to aid in the processing of data. Every single tool thus is responsible for data structuring, validation, and so on. The basic means to exchange data is in place, but nothing is yet known of the data's structure. In the control integration dimension, carrier level integration e.g. would mean to agree on using Remote Procedure Calls for exchanging control messages but saying nothing about the expected messages or parameter types being passed. In presentation integration, carrier level integration could mean fixing the use of the X11 windowing system without any restriction on the use of a particular widget set.
2. *Lexical level integration* is concerned with lexical conventions. In data integration, this could mean that all tools would use a common preprocessor on data files that resolves macro definitions and include files. Another example would be the convention to use a leading "." in text files to introduce a control command. Nothing is yet said about which control commands would mean what to a particular tool. In control integration, on the lexical level a certain format for control messages would be adhered to. This level of integration is for example achieved by Sun's ToolTalk without taking into account a specific set of message types. In presentation integration, on this level we would fix a certain widget set, e.g. Motif or OpenLook, without defining how these widgets should be arranged in a typical design tool.
3. *Syntactic level integration* is concerned with structuring conventions. On this level, tools agree on a set of data structures like parse trees in data integration, message categories like object creation, deletion, and change notification in control integration, or common application skeletons with a menu bar with "File", "Edit", and "Help" buttons on top, a scrolling window in the middle and a status line at the bottom. This is the level most commonly found in today's integrated design environments.
4. *Semantic level integration* goes a step further and defines a common understanding of data structures' semantics. This information can either be hard-

coded in the integrated tools or can be contained in a data dictionary for ease of reference and modification. With data integration, not only the syntactic structure of a design description has to be agreed on but also the semantics of the data. In control integration, a message dictionary would be set up which defines an agreed upon set of messages to be used. In presentation integration, a set of higher level, interface building blocks like complete graph flow browsers could be offered. Only proprietary, closed systems have reached this level of integration up to today.

5. *Method level integration* determines not only the kinds of objects which tools deal with, the kind of control messages they may exchange and the interface building blocks used in their construction, but fixes the use of the tools themselves within a methodology.

2.6 Granularity

The selection of an appropriate granularity is of key importance for design tool integration. Although this certainly holds for all integration dimensions, here we focus on data integration because all higher dimensions can profit from good data integration. As stated above, we may distinguish between the data handling mechanisms storage, transport, processing, and browsing. When looking at contemporary integrated design systems, we can distinguish three different granularities: *File-based, object-based,* and *value-based*.

File-based design data handling is the coarsest granularity found in integrated design systems today. It is dictated by file-based design tools. What is actually required by such tools, though, is file-based design data *transport* to and from the tools. The tools themselves *process* design data at the granularity of individual attribute/value-pairs. *Storage*, when concerned with version derivation, transactions, and equivalence relationships between individual objects, is most naturally done at the granularity of design objects. This is in line with the approach taken in the Nelsis CAD Framework [vanderWolf 90] and with earlier work (e.g. [Katz 87]). At this granularity, the framework stores information *about* the design objects as structural information, but does not need to make any assumptions about the actual detailed design representation. Assuming that tools handle the conversion from files to values in their input/output processors, our task is to easily convert from objects to files and vice versa, to bridge the granularity mismatch between object-based storage and file-based transport.

2.7 Integration approaches

When talking about tool integration, commonly only two extremes are distinguished, based on whether a tool's source code has to be modified to achieve integration or not:

Definition: *Black-box integration* or *encapsulation* assumes that the tool's source code contains no function calls to framework services. The tool only interacts with the native operating system services. All interaction of the tool with framework services uses a wrapper that translates tool data and control requirements to appropriate calls to framework services.

Definition: *White-box integration* or *tight integration* assumes that a tool is integrated into a framework by embedding function calls to framework services in the tool's source code. All interaction with framework services can thus be done directly by the tool.

Sometimes an intermediate integration approach *(grey-box integration)* is also considered [Kathöfer 90]. With this approach, the framework data-handling system emulates a file system and stores design data as unstructured objects. It offers the usual programming interface to tools, with functions like *open, seek, read, write,* and *close*. Like white-box integration, the tool's source code has to be modified to call these functions instead of the operating system's ones. Modifications are much simpler, though, because the file system functionality is closely emulated. While the data-handling system can offer extra protection of the design data by access control and transactions, like black-box integration, it has no knowledge of the internal structure of the design data and therefore cannot aid in management and browsing of design data structure. For this reason, we will not consider grey-box integration in the sequel.

Integration approaches are often associated with granularity of design data transport and storage. As encapsulated tools normally expect design data to be transported in files, both storage and transport are assumed to be file-based. We have reasoned above that file-based design data storage prevents effective management and browsing of design data structure. On the other hand, tightly integrated tools are assumed to transport design data across a programming interface in a value-based manner, therefore storage is often also assumed to be value-based.

Often, a distinction in control granularity is also assumed. Encapsulated tools are started and then left unattended until they either succeed or fail. Tightly integrated tools give access to more fine-grained functionality that may be invoked individually. This distinction may be true in most of the cases where the only way to control the functions a tool performs is through appropriate command line parameters. Nowadays, only few design tools are this uninvolved. Most provide a complex command language that can be used to control their actions. Interfacing to the command interface of a tool is desirable even if there is no programming interface to it, to be able to control the tool more flexibly.

2.8 Tool observability and controllability

Design tools manipulate design data by transporting them from a source, processing them, and transporting them back to a target*. Controllability deals with the question

> *"How and to what degree can a* program *invoke atomic operations in a tool."*

Observability deals with the question

> *"How and to what degree can a program tell which design data a tool reads and writes, and in what status it leaves them?"*.

It is important for integration that both kinds of interaction with a tool can be performed by programs and not just through the tool's user interface. The more observable and controllable a design tool is, the more tightly it can be integrated into a design environment. A tool is written to interact with a particular execution environment. For tools designed with a particular set of framework services (e.g. design data; data, methodology, and session manager) in mind, the execution environment comprises of the programming interfaces of these framework services. Most standalone tools, however, only rely on operating system services (e.g. file system, processes, terminals, window system). Here the execution environment comprises of the system calls offered by these operating systems services.

*. We have used the vague terms "source" and "target" here to account for the abundance of different ways to build the interface between a design tool and its design data, ranging from files managed by the operating system, streams passed between the tool and a framework or other tool through a socket or pipe, or even calls to a programming interface.

We now introduce two encapsulation approaches, a-priori encapsulation and tracing encapsulation, which differ in the amount of observability and controllability they assume of encapsulated design tools:

Definition: With *a-priori encapsulation*, a tool wrapper anticipates any possible request for an environment service by the tool in advance and prepares the execution environment accordingly.

The JESSI Common Framework, for example, ([Kathöfer 92], [JCF 94a]) assumes the following, fixed execution sequence to perform a design step:

1. The framework checks out a complete design 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 either success or failure.
5. The wrapper checks this result to decide on a number of post-processing alternatives, again processes and moves files and finally informs the framework of success or failure of the 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.

More flexible solutions may scan a set of working directories for new files or examine log files produced by the tool to deduce the actions taken by the tool.

Definition: With *tracing encapsulation*, a tool wrapper traces requests for environment services by any of a number of possible ways, logging notifications and servicing requests.

If the tool natively interacts with an operating system, the tool wrapper has to emulate operating system service requests issued by the tool (e.g. 'open design file X') and to translate them on the fly into the appropriate framework services calls. There are a number of ways to achieve this:

- In operating systems that support shared libraries, the *open* system call may be replaced with a routine that knows about framework services and redirects file requests to it. Design tools do not have to be recompiled but automatically use the new *open* routine when at start-up they dynamically load the system library containing it.

- Where this is not feasible, the *ptrace* system call allows to divert control at system calls to the wrapper at the cost of some execution time overhead.
- An altogether different approach does not try to manipulate the tool but rather lets the framework data-handling system emulate an NFS file system. The workstation at which the tool is running mounts this file system as usual and from then on design tools would see no difference between an ordinary file system and the one managed by the framework. The framework, on the other hand, has complete control over the design data read or written by the tool. It has to provide directory entries according to the design objects it manages. Whenever a tool opens one such object for read or write, the framework emulates file accesses on this object.*
- A last resort is to either trace log files created by the tool or regularly scan working directories touched by the tool to find out what happens.

A major problem with tracing encapsulation is that it is very tool-specific and can require much effort to realize. We can distinguish four cases of interaction of a tool and its execution environment (Table 3):

		environment	
		passive	active
tool	passive	**1**	**2**
	active	**3**	**4**

Table 3. *Cases of interaction between a tool and its execution environment*

1. With a passive environment, all tool/environment interaction has to be anticipated before the actual tool run and the execution environment has to be set up accordingly. Tool activities may be logged but not influenced. The classical way of encapsulating a design tool using a shell script can only result in this kind of interaction. Even with this low level of interaction impressive results can be achieved [Casotto 90].

*. While this approach seems to require much programming effort, the NFS protocol is publicly defined as internet standard [NFS], and the public domain operating system *Linux* contains an NFS server as starting point for an implementation [Linux].

2. An active environment intercepts environment interactions by the tools and can react accordingly. With this kind of interaction it is for example possible to check out additional design objects as files into the file system, start servers on demand, inhibit certain operations, or request user interaction.
3. An active tool requires that either the tool is designed from the start to issue logging calls to its environment or can be modified to do so. The programming interface of Casotto's VOV system, e.g. only requires the four functions *VOVbegin, VOVend, VOVinput,* and *VOVoutput* to be inserted into the tool's code.
4. The highest form of interaction can be achieved if both tool and environment are active. This way, the tool can actively issue requests to be serviced by the environment. A wrapper can translate operating system oriented requests into ones understood by the framework.

The problem with approaches (1) and (3) is that *all the 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.

If approaches (2) and (4) are to be combined with wrapper encapsulation, the standard UNIX shell as scripting language has to be replaced by a language more tailored to the task of tool encapsulation. We refer to the wrapper scripts as *tool execution protocols* to emphasize the fact that these scripts define a communication protocol between the designer, design tools, framework services, and the operating system. An execution protocol is comparable to a handshake protocol used in networking to define the state transitions of communicating agents. Figure 5 on page 37 shows an example 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:

Agent	Component	Method	Description
Designer		selectDO	use any of the means offered by the framework to select a design object
		select Activity	select an activity to apply to the design object
Frame-work	Framework tool	offerDO	offer a design object for manipulation
		offer Activity	offer an activity depending on the design object type
		notify ActEnds	present the result of the activity
	Design information manager	check-OutDO	check out a design object (flat/hierarchical) into the
		grantDO	notify the availability of a design object
		checkInDO	check in a design object from the file system
	Session manager	startTool	start a tool or connect to running tool
		startActivity	start an activity within a tool
Operating System	Process Manager		start/stop/signal tools
	File System		store files hierarchically

Table 4. *Methods implemented by communicating agents*

Agent	Component	Method	Description
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 contained design representation
		activityEnds	signal success/failure of the activity

Table 4. *Methods implemented by communicating agents*

Table 5 on page 36 depicts a sample execution protocol, written as messages exchanged between agents. 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. In this execution protocol, once an activity is started, it is in control to access design objects needed to complete its task. With this mode of operation, the design information manager has to provide access to design information and automatically keep track of objects in use to protect them against tool failure and conflicting concurrent access.

	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	notifyActivityEnds	DO activity

Table 5. *A sample execution protocol*

a. framework
b. design object
c. operating system

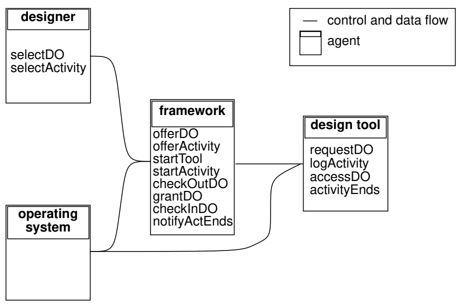

Figure 5. *Agents in a simple design system*

2.9 Related work

In this section, we will take a look at the literature on tool integration. It is widely recognized now that there are distinct dimensions to tool integration. Our dimensions *framework integration, data integration, control integration,* and *presentation integration* are based on work by Wasserman, who also developed the coordinate space diagram [Wasserman 90]. Thomas and Nejmeh refine Wasserman's original work with a focus on data integration [Thomas 92]. They identify five properties of data integration between the data management/representation aspects of two tools: *interoperability, nonredundancy, data consistency, data exchange,* and *synchronization* according to the following table:

persistent data	non-persistent data
interoperability	data exchange
data consistency	synchronization
nonredundancy	

Table 6. *Properties of data integration*

Two tools are well integrated with respect to interoperability and data-exchange if they require little work to be able to use each other's data. The distinction of persistent and non-persistent data is made because the mechanisms that support data-exchange integration may be different from the mechanisms that support interoperability integration. Two tools are well integrated with respect to data consistency and synchronization, if each tool indicates its actions and the effects on its data to other tools that might by affected. Again, the distinction of persistent and non-persistent data is made to account for different mechanisms. Two tools are said to be well integrated with respect to nonredundancy if they have little duplicate data or data that can be automatically derived from other data. Nonredundancy applies to both persistent and non-persistent data. The coverage of different dimensions for good integration is now also requested by standardization bodies ([ECMA 91], [CFI-FAR 93]). They focus, however, on the "core" dimensions data, control, and presentation integration.

Brown and McDermid add the dimensions *team integration* and *management integration* [Brown 92]. The prime contribution of their work, however, is their introduction of integration levels, which they discuss in the context of data and control integration with a focus on data integration. The justification for this restriction is the fact that "most IPSE's* have only tool integration†, and even there the integration potential is underused and the state of the technology is unsatisfactory." ([Brown 92], p. 25). Of course, this statement applies to software engineering.

Granularity is an important issue in recent work on tool integration. Researchers with a strong background in databases associate different levels with granularity, file-based granularity being the "lowest" and value-based or "fine-grained" granularity being the "highest" level [Kathöfer 90]. We rather agree with the more designer-oriented point of view that neither of these extremes make much sense for the end user of a design environment. The designer expects support by a framework and framework tools in the management of structural information, which is preferably managed at an object-based granularity [vanderWolf 90]. This view-point seems to be well supported by work done by Katz et al. at Berkeley, looking at both design data management and suitable user interface metaphors for design systems ([Katz 87], [Gedye 88], [Silva 93]). We know of no work that distinguishes between the granularities for storage, transport, and processing.

*. IPSE: integrated project support environment
†. tool integration: our data+control integration with a strong focus on data integration

The terms *encapsulation* and *tight integration* are common knowledge among researchers in the ECAD domain (e.g. [CFI-UGO 90]). A problematic aspect of this distinction is that it is often associated with integration granularity [Kathöfer 90], even more so because it is automatically assumed that file-based design data transport implies file-based storage. With tool encapsulation, there is some disagreement as to the amount of knowledge that should be provided about a tool in its tool abstraction specification. CFI captures only information relevant for tool invocation (tool name; executable pathname; argument type, name, and default values; return status; expected input and output data) [CFI-TES 93]. In the ULYSSES II System, Parikh et. al. require some more information to be able to select a tool suitable for a design task based on its abstraction [Parikh 93].

Tool control and observation is simple if a tool is tightly integrated into a framework and performs all the interaction with its environment through programming interfaces of framework services [Kupitz 92]. If a tool is not designed to operate in a framework environment and its source code is available, it can be enhanced to call functions from a small programming interface to mark actions relevant to design management [Casotto 90]. If, however, no source code is available, any of the encapsulation techniques described above has to be applied.

2.10 Conclusions

In the introduction we noted that the object-based management of structural design information separate from design representation information supports the designer's view of his design data. On the other hand, design tools clearly process design data on the granularity of individual values. Finally, transport to and from encapsulated design tools is mostly accomplished on the granularity of files. Value-based or *fine-grained* design data handling has often been advocated as a solution to this mismatch. We feel, however, that - from a design management point of view - framework supported, fine-grained design data handling is not feasible and, in fact, not desirable for the following three 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, object-based perspective biased towards structural information to provide execution contexts for its integrated tools. 'Coarse-grained' in this case is not to be confused with 'file-based'.

- As already pointed out in Section 1.5 on page 13, design structure tends to be domain independent information accessed by designers in ad-hoc, value-based queries. Design representation, on the other hand, is accessed by design tools by traversing the graph of related design objects, starting from some named or top-level design object.
- 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 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.

To conclude, we base our tool integration approach on the following

Three principles:

1. Design structure information is managed at the granularity of design objects.
2. Design representation information is transferred from and to design tools in design files.
3. For the purpose of this thesis, we assume design tools to be encapsulated.

Principle 1 guarantees that design data are managed (stored, browsed, and queried) at a granularity that is natural to designers and optimally matches the requirements for meaningful relationships like hierarchical composition, version derivation, and equivalence. Principle 2 matches the most popular way in which design tools interface with the outside world. This and principle 3 ensure that the source code of design tools need not be available to successfully integrate a design tool into an EDA environment.