Design Flow Management in the NELSIS CAD Framework

K.O. ten Bosch, P. Bingley and P. van der Wolf

Delft University of Technology
DIMES Design and Test Centre
Feldmannweg 17, 2628 CT Delft
The Netherlands

ABSTRACT

In this paper a new approach for design flow management is presented. We describe how the input/output relations between tools can be defined in a flowmap. For this, several concepts are introduced, such as defining activities for tools, run-time activity identification, hierarchical flow graphs, modification versus extension, and the possibility to have loops in the flowmap. We also address tool scheduling and the integration of design flow management in the architecture of a frame-based design system.

1. INTRODUCTION

Powerful integrated design systems are required for the design of complex electronic circuits. An essential base component of such a design system is a CAD framework, which serves as a basis for tool integration and provides the designer with assistance for data organization and design management.

Because of the increasing number of tools and variety of functions these tools perform, it becomes very difficult for a designer to learn about the available tools, their specific behavior, and the dependencies between tools. In addition, more advanced support is required for managing the status of the design and deciding on design steps to be performed.

It is for these reasons that it becomes necessary to extend frameworks with design flow management capabilities, which must be usable for a wide range of diverse tools. In this paper we present a concept for design flow management and how it fits in the architecture of a frame-based design system. The Nelsis framework, which is an open CAD-framework supporting versioning, hierarchy browsing, meta data management and easy tool integration, serves as an example. The concept of flow management however, is defined independent of the Nelsis framework and is therefore also applicable in other environments.

Section 2 describes some general requirements for flow management. One of them is the definition of dependencies between tools, which is described in section 3. A data schema for defining a flowmap is proposed in section 4. Section 5 and 6 are concerned with tool scheduling and some examples of the definition of a flowmap. Finally, in section 7 we will indicate how the flow concept can be implemented to yield powerful run-time capabilities.

This research was supported in part by the commission of the EC under project 5082 (Jessi-CAD-Frame).

2. FLOW MANAGEMENT

2.1 Requirements

Several authors [1,2,3,4] have proposed requirements for design flow management, from which we summarize the following list:

1. Flowmap configuration: the description of tools and dependencies between tools.
2. Tool scheduling: which tools can be invoked?
3. Automatic tool activation: tools which only perform a data transformation should be executed automatically.
4. Conflict resolution: which tool to choose when several possibilities exist to do the same task (e.g. 2 simulators).
5. Exception handling: when an error occurs in a design object the designer should be able to jump back to the state where this error was introduced (usually an editor).
6. Option and parameter selection: rules can be given to describe the parameters for each tool.
7. Methodology: design steps described in a general, tool-independent manner.

There are two aspects to executing design tools: performing a design task (for instance, simulate a design), or producing data (for instance, produce a plot file). In case a tool merely performs a transformation from one data format to another, it can generally be executed automatically and hidden from the user at the user interface level. It is the primary focus of a designer to successfully perform design tasks. Therefore it is more user oriented to have a flow concept dominated by design tasks to be performed, rather than data to be produced.

2.2 Configuration and Run-time Information

We divide the information concerned with flow management into configuration information and run-time information.

- *Configuration* information specifies the available tools and the relations between different tools. It is typically brought in by a design methodology manager before starting the actual design work. Configuration information is put in a *flowmap*.
- *Run-time* information is generated during the design process to correctly administer the state of design. It can be exploited to
  - monitor the consistency of design data,
  - inform the user about the state of the design, and
  - decide what tools can or should be run (tool scheduling).

The run-time information is updated after each tool run, while the configuration information changes only when tools are added, removed or replaced.

3. FLOWMAP DESIGN

Two ideas form the foundation of our flow concept regarding the description of tools and the dependencies between tools:

1. In frame-based design systems, data generated by one tool will be used as input for another tool. We model this

communication by treating tools as "functional units", which can be connected by "channels" to describe data transfer.

2. By describing the communication between tools in terms of abstract "datatypes" file details are hidden from the user. So the user may think in terms of "layout data", "test data", etc, while data of these datatypes is actually made up of several related files.

To describe more formally how a flowmap is built of functional units and channels, we introduce "ports":

Definition: Each functional unit can have ports. A port is either an input port or an output port of a functional unit and corresponds to a specific datatype. Channels connect ports of the same datatype.

From this definition it follows that it is possible that:

1. Several functional units need the same datatype as input, which is produced by one other functional unit.
2. A functional unit needs a specific datatype as input, which can be produced by one of several other functional units.

So a flowmap can have branches as well as merges, as can also be seen from the example in figure 1. This is in contrast with the approach of van den Hamer [5] where one input port can not be connected to several output ports. In that way a situation where layouts can be generated by several editors or synthesis tools, and used by another tool, can not easily be modeled.

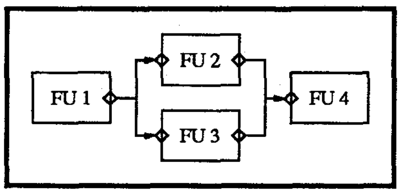

Figure 1. Flowmap with functional units (rectangles), ports (diamonds) and channels (arrows)

3.1 Activities

At first sight it is easy to design a flowmap that describes the data dependencies between tools. However, in practice the interaction between tools is more complex, since it is not always possible to know in advance how a tool will act, as tool behavior may vary due to several external factors, such as

- command line arguments, like options and parameters
- other parameters, such as process technology
- input data files
- user interaction.

We will refer to these four categories as "tool variables". Due to the influence of tool variables tools may produce data of different sets of datatypes for different runs. If these datatypes are not fixed for a particular tool it is unknown after a tool run if the appropriate data is available. Without this knowledge tool scheduling is impossible.

We solve this problem by removing the dependencies of tool variables. This is achieved by defining "activities" for tools such that for each activity its behavior is well defined.

Definition: An activity is a functional unit, which can only run when data is present for all its input ports and which produces data for all its output ports every time it runs.

A tool needs to be split into different activities if the input/output characteristics are different for these activities. On the other hand, if a tool does not act differently concerning its inputs and outputs, it may be split into activities, but there is no direct need to do so. Generally, an activity will correspond to a design function of a tool. However, it will be a decision of the flowmap-designer to split a tool into several different activities.

3.2 Hierarchy of Flow Graphs

Another valuable concept in defining a flowmap, is the concept of hierarchy. We introduce the term "flow graph" as a generalization of activity.

Definition: A flow graph is either an activity, or consists of several other flow graphs.

The flow graph hierarchy describes the instantiation of child flow graphs in their parent flow graphs. A flowmap is a flow graph without parents (the highest level in the hierarchy), activities are flow graphs without children (the lowest level). At intermediate levels all kinds of compound flow graphs can be constructed. The only constraint for this hierarchy is that it is a directed acyclic graph. The hierarchy depicted in figure 2a shows that flow graph C is instantiated in several other flow graphs, that flow graph E is instantiated more than once in the same parent flow graph and that several flowmaps can be defined in the same hierarchy (FM1 and FM2). Channels are used to connect ports of a compound flow graph to ports of its child flow graphs (see figure 2b).

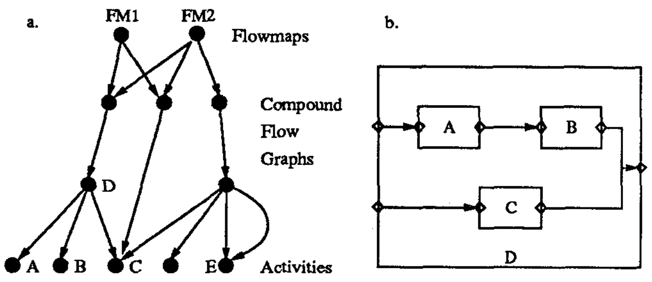

Figure 2. Hierarchy of flow graphs

The hierarchy concept is valuable in several ways. For instance, it is possible to define a "macro" by grouping a set of activities which must be executed automatically in a predefined sequence. Also, several similar activities may be instantiated in the same compound flow graph to clarify that they have almost the same function. Only one of them needs to be chosen by the user, or by an earlier specified personal preference. Summarizing, the hierarchy concept can be used to

- reduce the complexity at the user interface level by hiding details in compound flow graphs.
- improve project organization by defining compound flow graphs which correspond to high-level design tasks.

3.3 Properties of Ports

3.3.1 Input Ports

The definition of activity states that it consumes all its input datatypes and produces all its output datatypes. Although this concept is powerful enough to model all possible situations, we introduce "optional" input ports for convenience, to prevent an activity split when an input datatype is optional.

Definition: When an activity has an optional input port, it can run with or without data of the datatype corresponding to this input port. An input port which is not optional is "required".

3.3.2 Output Ports

Many frameworks offer a version mechanism to support the existence of different versions of the parts of a design. For instance, in case of an editor modifying a design object, a new version of this object is created. To support versioning it is necessary to distinguish between two types of output ports:

modification ports and *extension* ports. By extension is meant that the produced data is stored in an existing design object. For example, an expanded netlist may be stored with the corresponding schematic. In the case of modification a new design object is created for storing the produced data.

When an activity generates data via an extension port, while data of the same datatype already exists for the design object, the existing data should be invalidated. For modification ports data is never overwritten since it is stored in a new design object. So modification ports never invalidate existing data. For extension ports we define the following invalidation rule:

Invalidation rule: When an activity produces data via an extension port while data of the same datatype already exists, the existing data and all its derived data produced via extension ports becomes invalid.

3.4 Loops

It is often necessary to define loops of activities in a flowmap. However, we must prevent that the activities in a loop can never be executed (deadlock). It is useless to define a loop (a cyclic sequence of activities) in a flowmap, which has no starting point, because in that case no activity of the loop can ever be executed. The starting point of a loop is an activity which can be activated when no other activity of that loop has been executed. This results in the following definition:

Definition: A loop of activities is allowed if it contains at least one starting point, that is, an activity for which each input port is either an optional port or a port connected to an activity not contained in the loop.

In figure 3, situation 1 contains a forbidden loop because the activities can never be executed. The loop in situation 2 is allowed since activity A can fill port IP1, thereby providing a starting point for the execution of the activities in the loop. Optional ports are represented by solid circles.

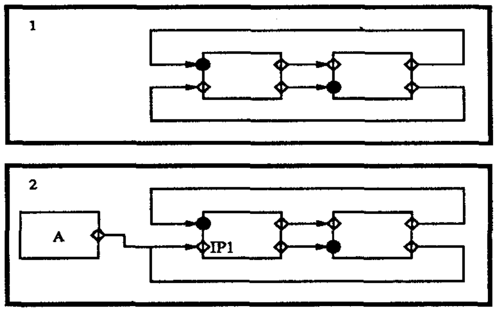

Figure 3. 1 contains a forbidden loop, the loop in 2 is allowed

Since loops are allowed in a flowmap, we have to refine the invalidation rule. Consider the situation as shown in figure 4. According to the invalidation rule, a run of activity A would invalidate the data corresponding to the output ports OP1, OP2, OP3 and OP4. So it would invalidate its own output data. The invalidation rule is refined as follows:

Invalidation rule: When an activity produces data via an extension port while data of the same datatype already exists, the existing data and all its derived data produced via extension ports becomes invalid, except for data produced by the activity itself.

3.5 Design Constraints

There is a relation between channels, input ports and freedom of design. When input ports are added to activities, data of the corresponding datatypes must be present in order to run the activities. This specifies constraints. When channels are added to

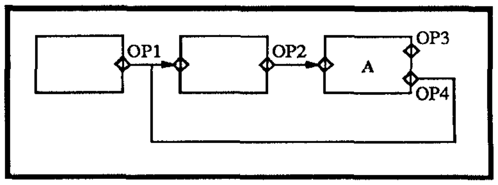

Figure 4. A invalidates OP1 and OP2

connect ports, these specify the possible ways to produce data of the needed datatypes. This specifies a relaxation of constraints. Because it is the responsibility of the flowmap designer to add the appropriate channels and ports, this gives him the opportunity to specify constraints that are not really necessary from a datatype point of view. This can be used to implement for example, an electronic signature. In this way design constraints, which should otherwise be specified as external rules, can be defined as part of the flowmap. An example of the specification of additional constraints is given in section 6.

4. THE DATA SCHEMA

In Nelsis the OTO-D data modeling technique [6] is used to define a data schema for the meta data (data about the design data) [7]. It is possible to use this method also to define a data schema in which the flowmap can be defined. From the discussion in the previous sections follows that this schema should at least contain object types such as, activity, flow graph, port, channel and datatype. We will briefly explain the most important object types in the data schema of figure 5.

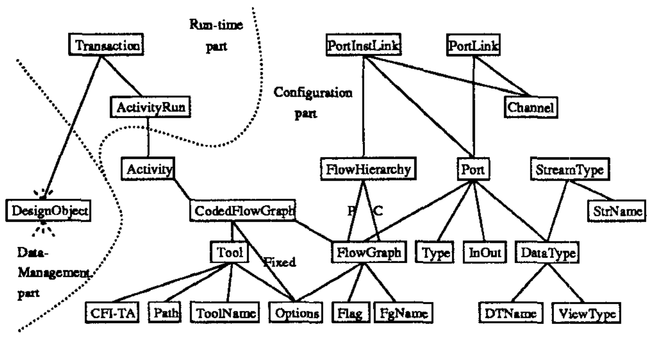

Figure 5. Data schema for defining a flowmap

4.1 The configuration part

Flow Graph: Either a leaf flow graph or a compound flow graph. A compound flow graph may be a coded flow graph. A flow graph may be defined to be automatic by the attribute *Flag*.

Coded Flow Graph: A flow graph which directly corresponds with a tool. This tool may perform an arbitrary number of activities related to this tool, or invoke other tools which perform their activities.

Activity: A coded flow graph, for which the input/output characteristics are known in advance. Activities are always leafs in the flow hierarchy and leaf flow graphs are always activities.

Tool: An executable program, characterized by its *Path*, *ToolName*, *Options* and *CFI-Tool Abstraction* attributes [8].

FlowHierarchy: This type is used to instantiate Child flow graphs in Parent flow graphs, i.e. to define a hierarchy of flow graphs.

Port: A flow graph can have several ports. Each port is either an input port or an output port, specified by the attribute *InOut*, and corresponds to a datatype, specified by *DataType*. The *Type* of a port specifies for an input port whether it is required or optional and for an output port whether its type is extension or modification.

DataType: Set of data to be handled together. A datatype belongs to exactly one view type.

StreamType: A datatype contains no, one or several streamtypes. A streamtype belongs to exactly one datatype.

Channel: Data transport channel for transporting data of one specific datatype. A channel is connected to ports by a number of producer links (at least one) and a number of consumer links (at least one).

PortInstanceLink: A PortInstanceLink connects a channel to a port of an instantiated flowgraph. If this port is an input port the link is a consumer link otherwise it is a producer link.

PortLink: A PortLink connects a channel to an uninstantiated port. If this port is an input port the link is a producer link otherwise it is a consumer link.

4.2 The run-time part

ActivityRun: When an activity has been executed this is administered in the run-time part of the meta data. Besides the activity that has been performed, the activity run has additional attributes (not shown) like date, time, user and used options.

Transaction: An activity may work on certain design objects and may produce certain design objects. A transaction relates design objects to activity runs in such a way that an activity run may be related to several design objects, and one design object may be involved in several activity runs.

4.3 The data-management part

Design Object: The design object is the central object type in the data management part. For a description of the data management part of the data schema we refer to [7].

5. TOOL SCHEDULING

The flow manager should specify at each stage in the design process which tools can be invoked. First of all we define the semantics of "invoking" a flow graph.

Definition: *Invoking* a flow graph causes
- the corresponding tool to execute with the fixed options, if the flow graph is a coded flow graph.
- to invoke several child flow graphs based on an invocation algorithm, in other cases.

The goal of such an invocation algorithm will be to fill as many output ports of the flow graph as possible. Since there may be several ways to fill all output ports, due to parallel paths, it is useful to define "execution paths" of a compound flow graph:

Definition: An execution path of a compound flow graph is a sequence of child flow graphs which, when executed, fill all its output ports.

For example, the compound flow graph D in figure 2b has two execution paths (executing A and B, or executing C). For an arbitrary flow graph the executable execution path(s) can be searched for by a simple algorithm which traces recursively from the output ports to the input ports. In the case where several different execution paths exist, one could be chosen according to a specified preference, or the user could be asked to make a selection. Now that we have defined the invocation of a flow graph, we give suggestions of how a user interface can show whether a flow graph can be invoked.

Definition: A flow graph is *executable* iff
- all required input ports are filled, if the flow graph is an activity.
- all its output ports can be filled, in other cases. When it has no output ports, it is executable if all its children are executable or have executed.

Because an activity fills all its output ports when it runs, there is a strong equivalence between both cases. Note that a compound flow graph is either not executable or completely executable. For increased designer assistance we would like to know whether a compound flow graph can be partly executed by executing one or more of its children. Therefore we define:

Definition: A flow graph is *partly executable* iff
- it is executable,
- or it has at least one child which is partly executable.

We also have to define when a flow graph has executed:

Definition: A flow graph has *executed* iff
- an activity run is administered and its output data has not been invalidated, if the flow graph is an activity.
- all its output ports are filled, in other cases. When it has no output ports, it is executed when all its children have executed.

Similar to the definition of partly executable, we define:

Definition: A flow graph has *partly executed* iff
- it has executed,
- or it has at least one child which has partly executed.

These definitions provide a basis for implementing tool scheduling, to assist the designer in selecting design tasks to be performed. When these mechanisms are used in the right way, they can become powerful instruments.

6. FLOWMAP EXAMPLE

In this section we will describe some possibilities to define a flowmap for the checking and simulation of a layout. Suppose that our design environment consists of the following tools:

- LAYOUT EDIT: a layout editor
- EXP: expands a layout description
- CHECK: a layout design rule checker
- EXTRACT: extracts a circuit description from a layout
- SIM1, SIM2, SIM3: several simulators

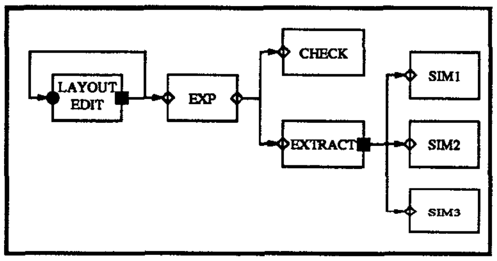

Figure 6. A flowmap for layout design

In order to run either CHECK or EXTRACT on a layout, EXP must have been executed. Simulating a layout can only be done when the layout is extracted. If no compound flow graphs were used, the flowmap would look like figure 6. A modification port is represented by a solid square.

However, it is desirable to present the flowmap to the user as simple as possible. So some related tools can be combined into a compound flow graph. For instance, a compound flow graph consisting of the three different simulators can be defined. Other compounds DRC and EXTR can be built from EXP and CHECK, and EXP and EXTRACT respectively. This would produce a flowmap as shown in figure 7. The internals of the compound flow graphs DRC, EXTR and SIM are also shown.

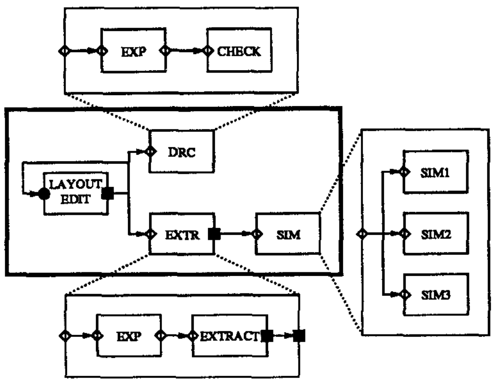

Figure 7. A flowmap with compound flow graphs

Note that the activity EXP is instantiated in two different compound flow graphs. At the user interface level, this means that at any moment both will have the same state and when EXP in DRC is executed, both CHECK and EXTRACT become executable. Because data management is done by the framework kernel, this is handled correctly.

As described earlier, additional design constraints can be specified in the flowmap. For instance, the flowmap designer may wish to forbid the extraction of a circuit from a layout, before the layout has been checked. This is described by creating channels to connect an output port of DRC to an input port of EXTRACT, as shown in figure 8.

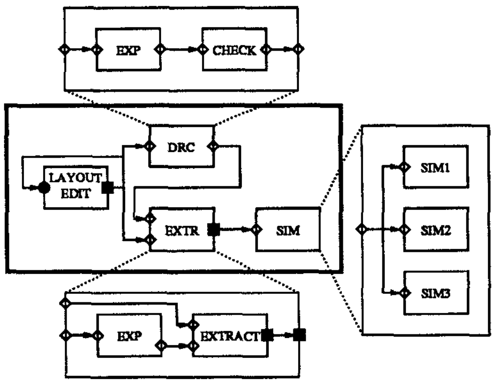

Figure 8. DRC must be executed before EXTRACT

7. IMPLEMENTATION

7.1 Introduction

The presented design flow concept is claimed to be completely general, in the sense that it does not contain any assumption about a particular design environment. However, successful application of the design flow concept strongly depends on the way it is incorporated in the architecture of a design system. In the following sections, we will describe some implementation choices. The Nelsis IC Design System serves as an example.

7.2 The Nelsis Architecture

The architecture of the Nelsis IC Design System [9] is presented in figure 9. It shows that tools, which may be either encapsulated tools or tightly integrated tools, are integrated on top of a framework. The framework provides various kernel services,

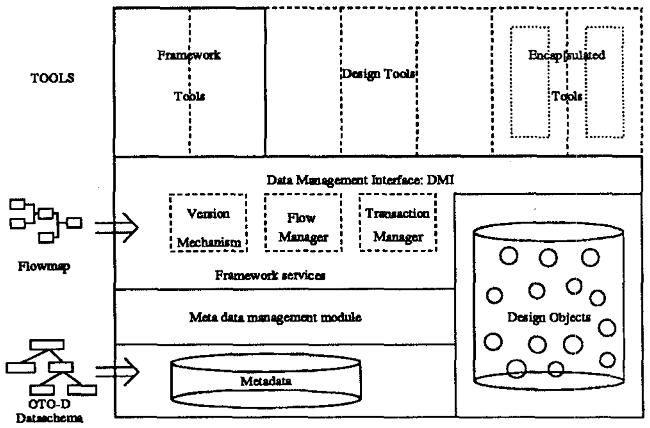

Figure 9. The Nelsis IC Design System architecture

such as version management and design transaction management, implemented on top of a high level meta data management module [10]. This module can be configured by a data schema, such as the data schema presented in figure 5.

7.3 Implementation Alternatives and Activity Identification

The presented flow concept can be implemented in a CAD system in various ways:

1. as a desktop-like framework tool only, which displays the flowmap, handles user interaction, fires tools and maintains the status of the design.
2. as a kernel framework service together with a desktop-like framework tool as the user interface. The kernel service monitors data accesses, to identify and correctly administer activity runs. The administration is made available to the user interface to display flows, present statuses and give advice on tasks to be performed. Tools can then be started from this user interface (but this is not necessary).

To decide on one of these alternatives we have to take the run-time identification of activities into account. As described in section 3.1 we permit multiple activities to correspond to a single tool. When tool behavior depends on command line arguments, parameters or input files, the activity can be identified *before* the actual tool run. A desktop will then be able to identify the activity by interpreting the tool variables. However, when tool behavior depends on user interaction, a desktop can identify the activity only *after* the actual tool run, if at all. Interactive tools can perform complex tasks on multiple design objects, under control of the designer. For these tools it is often not possible to identify the activities by inspection of output data after a tool run. Hence, only a desktop does not suffice. Since a kernel framework service monitors data accesses while they are being performed by the tool, it can identify and administer activities at run-time, provided that the following constraint is satisfied:

When a tool implements several activities, these activities must be recognizable by datatypes or command line arguments.

7.4 Implementation of Flow Management in Nelsis

Tools interact with the framework via the Data Management Interface (DMI) [11] to obtain access to design data. By offering a proper set of "anchor points", the standard DMI greatly facilitates software exchangeability and permits framework and tools to evolve separately to a large extent. As an illustration of how the DMI functions cooperate, we present a calling pattern in figure 10. Besides cell names, modes, etc. such information as view type, tool name, and streamtypes is passed as arguments via the DMI functions.

```
DM_ENVIRON envkey;
DM_PROJECT projectkey;
DM_CELL cellkey;
DM_STREAM streamkey;

envkey := dmInit (toolname);
  projectkey := dmOpenProject (envkey, projid, openprojmode);
    cellkey := dmCheckOut (projectkey, cellid, checkoutmode);
      streamkey := dmOpenStream (cellkey, streamid, iomode);
        dmPutDesignData (streamkey, format, arguments);
        dmGetDesignData (streamkey, format, arguments);
      dmCloseStream (streamkey, closestreammode);
    dmCheckIn (cellkey, checkinmode);
  dmCloseProject (projectkey, closeprojmode);
dmQuit (envkey);
```

Figure 10. DMI calling pattern

By matching the information obtained via the DMI against the configuration information as modeled in the data schema, activities can be identified and activity runs can be administered. In Nelsis, design data is organized as *design objects*, and per design object data is organized as one or more *streams*. For example, streams may correspond to design files. Since each datatype corresponds to a number of streamtypes, as administered in the data schema of figure 5, activity identification can be based on stream accesses performed by the tools.

Hence our implementation of the presented design flow concept can handle encapsulated tools being run from a desktop, as well as tightly integrated tools which at run-time determine the activity to be performed. For example, it can handle interactive editors that may perform multiple edit operations during a single run, with cells being selected interactively by the designer. Also, it can handle tools whose actual behavior is controlled by a command file which is interpreted at run-time. Desktop systems can handle this situation only by interpreting the command file prior to the actual tool-run, which is cumbersome and inefficient.

8. RELATED WORK

In the HILDA environment [1] a mechanism based on Predicate-Transition Petri nets is used to specify tools and their interactions. Production rules with automatically updated certainty factors, are used to give designers additional help.

The ULYSSES [3] environment uses a blackboard architecture for the communication among CAD tools. A scheduler selects the appropriate CAD tool based on current design goals. A language is used to define tasks, subtasks and consistency rules.

Van den Hamer [5] defines a data flow based architecture for CAD frameworks. It includes the description of dependencies between tools together with a version mechanism which supports derivation, modification and regression.

In contrast with these other approaches, which implement flow management as a desktop-like tool or specify nothing about the implementation in a framework, we presented a concept to fit design flow management in a frame-based design system. Furthermore, by introducing activities, we specified how to cope with tools that do not act straightforward at all with respect to flow management. This makes our concept usable for a wide range of diverse tools, while some of the other approaches have difficulties in handling, for example, interactive tools. In our approach the flowmap contains reliable information, which can be used to reflect the exact state of the design at any moment. This has enabled us to give definitions for tool scheduling, which can be used to build a user interface for designer assistance.

9. CONCLUSIONS

In this paper a concept for design flow management in an integrated design environment has been presented. It incorporates a powerful concept for describing design flows, based on a general approach, to minimize the restrictions on tools to be incorporated. Design flows can be defined in terms of activities and relationships between activities. A single tool is allowed to perform one of many possible activities. Hierarchical design flows are supported and loops are allowed under well-defined conditions. Run-time identification and administration of activities is handled in the most flexible way by implementing the concept as a kernel framework service.

Definitions of the executability of tools have been presented. The user interface of the flow manager can exploit the presented flow concept to combine data browsing capabilities, status display based on flows, tool scheduling advice and automatic tool activation.

We are currently upgrading a first prototype, implementing the presented concepts, for incorporation in existing user interfaces. In addition we are looking for solutions to the problem of conflict resolution, which occurs when there are parallel paths in the flowmap. We are also investigating ways to incorporate design methodology management, by extending design flow management with concepts for management of design policies and strategies.

References

1. F. Bretschneider, et al., "Knowledge Based Design Flow Management," *Proc. ICCAD - 90*, (1990).
2. J. Daniell and S.W. Director, "An Object Oriented Approach to CAD Tool Control Within a Design Framework," *Proc. 26th DAC*, pp. 197-202 (June 1989).
3. M. Bushnell and S.W. Director, "Automated Design Tool Execution in the Ulysses Design Environment," *IEEE Trans. on Computer-Aided Design* 8(3)(March 1989).
4. A. Di Janni, "A Monitor for complex CAD systems," *Proc. 23rd DAC*, pp. 145-151 (1986).
5. P. van den Hamer and M.A. Treffers, "A Data Flow Based Architecture for CAD Frameworks," *Proc. ICCAD - 90*, (1990).
6. J.H. ter Bekke, "Database Design (in Dutch)," *Stenfert Kroese*, (1988). English version: "Semantic Data Modelling", Prentice Hall, ISBN 0-13-80605 0-9.
7. P. van der Wolf and T.G.R. van Leuken, "Object Type Oriented Data Modeling for VLSI Data Management," *Proc. 25th DAC*, (1988).
8. CFI Design Methodology Management TSC, "Tool Abstraction Specification (Draft Proposal)," *Document 51, Version 0.21*, (February 5, 1991).
9. P. van der Wolf, P. Bingley, and P. Dewilde, "On the Architecture of a CAD Framework: The NELSIS Approach," *Proc. 1st IEEE European Design Automation Conference*, (1990).
10. P. van der Wolf, G.W. Sloof, P. Bingley, and P. Dewilde, "Meta Data Management in the NELSIS CAD Framework," *Proc. 27th DAC*, (1990).
11. N. van der Meijs, P. van der Wolf, I. Widya, and P. Dewilde, "A Data Management Interface to Facilitate CAD/IC Software Exchanges," *Proc. ICCD '87*, (1987).