Design Flow Management in CAD Frameworks

Olav ten Bosch

Delft University of Technology
September 1995

Design Flow Management in CAD Frameworks

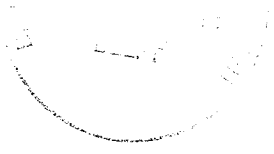
PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op maandag 2 oktober 1995 te 10.30 uur

door

Karel Olav TEN BOSCH

informatica ingenieur
geboren te Leiden



Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. P.M. Dewilde

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof. dr. ir. P.M. Dewilde, TU Delft, promotor
Prof. dr. ir. R.H.J.M. Otten, TU Delft
Prof. dr. W. Gerhardt, TU Delft
Prof. dr. P. Marwedel, TU Dortmund, Duitsland
Prof. dr. S.W. Director, CMU Pittsburgh, USA
Dr. ir. P. van der Wolf, TU Delft

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bosch, Olav ten

Design flow management in CAD frameworks / Olav ten Bosch.
- Delft : Delft University of Technology. - I11.
Also Thesis Technische Universiteit Delft, 1995. - With
ref. - With summary in Dutch.
ISBN 90-5326-021-8
Subject headings: CAD frameworks.

CONTENTS

1. Introduction 1
   1.1 Design Complexity 1
   1.2 Design Flow Management 4
   1.3 Initial Approach and Requirements 8
   1.4 Origination of Our Design Flow Paradigm 9
   1.5 Overview of this Thesis 10

2. Design Flow Management 11
   2.1 Introduction 11
   2.2 CAD Frameworks 12
   2.3 Design Flow Management Approaches 16
   2.4 Global Approach 21
   2.5 Conclusion 34

3. Tool Modeling 35
   3.1 Introduction 35
   3.2 Modeling Data Access 37
   3.3 Modeling Tool Control Information 47
   3.4 Comparison with Other Approaches 53
   3.5 Conclusion 55

4. Design Flow Modeling 57
   4.1 Introduction 57
   4.2 Modeling Dependencies 58
   4.3 Design Flow Hierarchy 69
   4.4 Port Relationships 76
   4.5 Comparison with Other Approaches 79
   4.6 Conclusion 81

5. The Design Flow User Interface 83
   5.1 Introduction 83
   5.2 Flow-Based User Interaction 85
   5.3 Flow Coloring 92
   5.4 Design Scheduling 115
   5.5 Comparison with Other Approaches 123
   5.6 Conclusion 125

6. Implementation in Nelsis 127
6.1 Introduction 127
6.2 The Nelsis Tool Model 128
6.3 The Nelsis Design Flow Model 141
6.4 The Nelsis Design Flow User Interface 144
6.5 Changing Design Flow Configurations 148
6.6 Design Flow Management in the Nelsis Architecture 151
6.7 Related Issues 160
6.8 Examples 163
6.9 Conclusion 175

7. Conclusion 179

A. Appendix 183

References 189

Samenvatting 197

Acknowledgements 201

Curriculum Vitae 203

1

Introduction

1.1 Design Complexity

This thesis is about assisting designers in complex computer aided design (CAD) processes. To explain this, we compare the activities of a designer with driving and operating a car.

The total knowledge and skills required to move on by car from one place to another are quite extensive. First of all, one must have learned the basics of car driving. Although each of the individual actions like using the break, putting on the direction indicator, switching on the lights, looking into the mirrors and turning the steering wheel are essentially very simple, to do these things in the right order and in the right situation is a complex task, which demands a lot of experience. Second, a driver must be aware of the particular traffic regulations in the areas where he drives. These regulations may contain rules like "when entering a built-up area reduce your speed to 50 km/hr at most". Third, a driver must somehow find out how to reach his destination. If the route is well-known to the driver, he may rely on his memory for this. If this is not the case, he will have to consult a map or (if available) read a signpost. In both cases, the driver "learns" about the topography of the roads. Finally, a driver must take other drivers into account. Unexpected traffic diversions, accidents or traffic jams may force a driver to choose another route.

There are many similarities between driving a car and operating a set of tools in a CAD system. First of all, a designer should have learned how to use the available design tools. Although individual design tools may be relatively simple to operate, navigating the entire design process is usually a complex task, which can only be practiced effectively by people with knowledge of the design environment and with enough design experience. Second, a designer should be aware of the particular design regulations in this particular design environment. For instance, it may be prescribed that a certain design operation is performed only if the design satisfies a number of specific conditions. Third, a designer must somehow find out

how to fulfill the end-goals. If the designer has performed this design task many times, he may rely on his experience to perform the right operations at the right times. If this is not the case, he will have to consult a manual or (if available) consult some local expert. In both cases, the designer "learns" about the topography of the local design environment. Finally, since in complex design processes multiple designers may work on related pieces of the same design, they must take the activities of their colleagues into account. Occasionally, they may have to re-plan certain intended design activities.

Although the use of the car-driving metaphor gives a general feeling of the type of problems a designer encounters, actually the complexity of a computer aided design process may exceed the complexity of driving a car by far. This is due to the following:

- In extensive design environments, the number of design tools may be large and design tools may have many modes in which they can operate, specified by command line options or environment variables. In this case it is no longer straightforward for a designer to know what to do in which situation. The number of applicable methods is simply too large to expect a designer to always take the right decision.
- In contrast to the skill of driving a car, which has not been changed for many years, new design tools are developed continuously and existing design tools are updated frequently. This makes that designers have to keep their knowledge of the operation of the design tools up to date.
- The huge amount of design data involved in complex design processes and the large number of redesigns that originate from experimentation and iteration make it difficult for a designer to select the right components for his design activities. Without mechanisms for an effective organization of the design data, a designer may easily get lost in his own design process.
- In the driver metaphor, the place of destination is a uniquely defined point in a two- (or three-) dimensional space. However, the situation in which a designer is situated is more like a driver who tries to find a place that satisfies a number of competing requirements and that he does not know to be located at all. A designer has to maneuver through a multi-dimensional design space to find a solution that satisfies a possibly large set of specifications.
- The number of people involved in a complex design process may be large. To exclude conflicting design actions, mechanisms and policies to guard the consistency of the design need to be offered.

The large number of ever changing design tools, the tremendous amount of design data, the lack of a unique design solution and the large number of people involved make the design process considerably complex. The increased *design complexity* calls for mechanisms to assist designers in executing their complex design tasks. It is the goal of this thesis to develop such mechanisms for the electronic design process.

In the case of driving a car many services exist to help drivers reach their place of destination. This begins with the help of a driving teacher, who explains the operation of a car and who teaches the local traffic regulations, which are being enforced by the local police. Road maps and signposts inform drivers about their route and finally the traffic news informs drivers via the car radio about accidents and traffic jams.

Borrowing from the driver metaphor, in this thesis we aim at developing means to guide designers through the design process by offering them a combination of a 'driving teacher', a set of 'signposts' and a 'radio traffic service'. Of course, in CAD these services can be automated. This means that each designer has his own personal 'driving teacher', who knows about the particular interests of the designer, that the 'CAD police' controls all design actions performed by all designers (it does not control just by snap checks), that, wherever the designer is located, he will always see a 'signpost' pointing in a direction that brings him closer to a desired design solution, and the 'CAD traffic service' will only give him messages that are of his interest.

The benefit of developing such facilities for electronic design is to increase the overall *design productivity*. The less time designers have to spend on organizational issues, the more time they have to focus on the specifics of the design under construction. Not only does this speed up the design process, it may also lead to a better design quality. Kleinfeldt et al. [Kleinfeldt94] conclude that the productivity improvements in this area are potentially large, but also that critical research is still needed. This thesis is supposed to add to this critical research. We study methods and tools that assist designers in performing complex design tasks in complex computer aided design processes. In the next section we start with an exploration of the fundamental problems involved and from there we proceed towards an initial solution to these problems.

1.2 Design Flow Management

Because of the large collections of design data involved in electronic design, traditionally the bottleneck in electronic CAD (ECAD) was a matter of data organization. Since existing data base techniques did not satisfy the specific needs for handling the large collections of interrelated design data, mechanisms had to be developed to support ECAD-typical constructs, like hierarchical design descriptions, multi-view representations and long duration transactions. In addition, to guard the consistency of design data in a multi tasking environment, methods for consistency control and access control were developed. Since these facilities had to be shared by all design tools in a design environment, this led to the origination of so called *CAD frameworks* [Barnes92, Wolf94a]. A CAD framework is a software infrastructure that provides a common operating environment for CAD tools [CFIugo90].

These days, CAD frameworks effectively solve most of the data management problems in CAD. Therefore, the bottleneck in ECAD is no longer a matter of data organization, it has moved to the organization of the design process itself. The increased design complexity makes it more difficult for designers to keep a good overview of the design process, which is necessary to perform their complex design tasks well. To be able to freely move around in the design space, trying different design solutions for each design problem, they need to be informed about the structure of the design environment in which they have to operate. They should have a clear idea about the available *design resources* (i.e. the design functions available) the *design constraints* (i.e. the restrictive relationships between the different design functions) and the *design objectives* (i.e. the specifications that must be satisfied) in a specific design process. In other words, they need to be informed about the ins and outs of the design process.

In any design process there is a notion of how to realize the end-goals of the design process, although this may be nothing more than an informal idea residing in the designer's mind. This notion usually indicates the different design phases and the order in which these design phases should be executed. Such a notion is widely known as the *design flow* of a design process (sometimes called a *design methodology*) [JCFgloss92]. In general, a design flow for a computer aided design process consists of a *specification* phase, a *synthesis* phase and an *analysis* phase (see for instance [Post87] and [KweeChrist95]). This is shown in figure 1.1. The design process starts with an idea residing in the designer's mind. This idea is transformed into a formal design *specification* that contains a more specific description of the problem and the requirements for a possible solution. In the *synthesis* phase a proposed solution to the design problem is built. This design

solution is analyzed in the *analysis* phase. Based upon the results of the analysis the specification can be refined or another design solution can be tried. This process proceeds until a satisfactory solution is obtained.

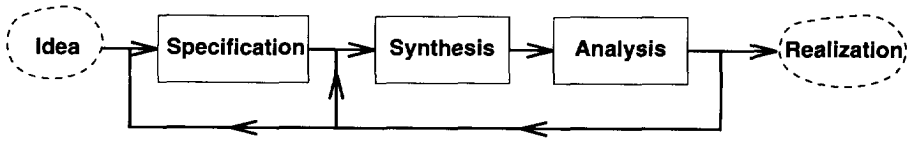


Figure 1.1. General design flow of a design process.

The above design flow is a very general abstraction of a design process. It recognizes only a limited number of individual design phases. In many design applications a much more specific design flow can be specified. This is particularly true for electronic design where, due to the large number of design tools, data formats, levels of abstraction and design constraints, designers need to remember an awful lot of details about the structure of the design process. Since all design operations performed within a design process are performed through the operation of design tools, a very detailed design flow may actually map the design phases of figure 1.1 onto design functions directly available to the designers. In the extreme, a design flow may reflect the complete structure of a design process, expressed in the design functions offered by the different design tools.

Since such a detailed design flow reflects the structure of a design process in a way that closely allies to the way designers experience this design process, we think it is a useful instrument to assist designers in mastering the complexity of their work. We want to exploit the notion of a design flow to equip a design system with detailed knowledge of the design process. For this, we have to transform the notion of a design flow from an abstract idea residing in the designer's mind into a formal specification used by a design system to assist designers with their complex design tasks. The net-result we are aiming at could be described as a *smart design system*, which not only enables designers to perform different design operations, but also knows about the specifics of the current design process and exploits this knowledge for improved designer assistance.

To actually use the notion of a design flow to make design systems smarter we define a design flow more formally. We like the definition of a design flow as used by a design system to correspond as closely as possible to the idea of a design flow as experienced by the designers. From the above discussion we conclude that a design flow describes how to go from design specification to realization and also that it contains a description of the available design resources, design constraints and design objectives. Therefore we define:

Definition 1.1:

A *design flow* is a description of a design process that specifies how to go from design specification to realization, taking into account the available design resources, the design constraints and the design objectives.

We use the term design flow to refer to the totality of information describing a particular design process. This information varies from descriptions of basic design tool capabilities to descriptions of complex design tasks, in which multiple design tools participate. For instance, a design flow may describe tools, tool characteristics, dependencies between tools, tools in relation to data, tool invocation characteristics, sequences of tools that form an abstract design task, how tools can be combined to reach a certain goal and a description of the design guidelines and strategies for a specific design process. We explicitly state that, although we speak of one design flow, this does not mean that a design flow defines exactly one fixed strategy. A design flow may be defined as strict or lenient as necessary.

Now that we have introduced the notion of a design flow, we can more specifically present our approach to building smart design systems. We expect a smart design system to utilize a design flow to assist designers during their work. In other words, we want such a design system to *manage* a design process based on a design flow. We capture this functionality under the name *design flow management*[1]. Before we give a definition of design flow management, we first inspect how it relates to the notion of a CAD framework. Above, we mentioned that a CAD framework provides a common operating environment for CAD tools. Such a common operating environment comprises a number of general services that are used by the different design tools in a design system. Since the smartness of a design system should benefit to all design tools and all designers in a certain design environment, in our view design flow management should be one of the services offered by a CAD framework. This brings us to the following definition:

Definition 1.2:

Design flow management is a set of CAD framework functions concerned with assisting designers during the design process based on a design flow.

1. Note that design flow management is *not* the management of a set of design flows, but rather the management of a design process *based on* a design flow. Since the term design flow management is widely used for this purpose, we take this possible source of confusion for granted.

We call that part of a CAD framework in which the design flow management functions are located a *design flow (management) system*. Roughly speaking, we distinguish between the following functions of design flow management:

1. *design tracking*:
   A CAD framework, controlling the access of the tools to the design data, can *track* a design process in terms of the data accesses performed. Although this does already provide designers with useful information on the history of the design process, a design flow system, having a description of the design process in a design flow, can do even better. It can *track* the design process in terms of the basic elements of the design flow and present the tracked design state in relation to this design flow. Such a flow-based representation of the design state helps designers to determine the progress they made and to decide on consecutive design operations to be performed.

2. *constraint enforcement*:
   As mentioned before, in complex design processes the number of design constraints designers have to remember may be large. A design flow system, knowing the structure of the design process, may *enforce* such design constraints and inform designers about possible and impossible design operations. In this way, designers can be kept from making certain design errors.

3. *design assistance*:
   Knowing the detailed structure of the design process, a design flow system can *assist* designers in many ways. First of all, instead of representing design operations as black boxes, as is usually the case in systems without design flow management, they can be given a representation that more closely reflects their essential properties. This gives designers a better feeling of the use of the different design operations. Second, instead of representing design tools and data as isolated entities, they can be integrated into one single natural representation of the design process, which is based on the design flow. The gain of using such a flow-based representation of the design process, is that designers can more easily determine what needs to be done next. Third, the invocation of a CAD tool can actually be quite complex. Especially in design systems comprising a large number of design tools, which each have multiple options and parameters, it can be difficult for designers to remember the proper tool invocation command for each tool. A smart design system, having detailed knowledge about the characteristics of the design tools, can assist designers during tool invocation, taking into consideration the current design state.

4. *design automation*:
   Knowing the structure of the design process, a design flow system may *automate* certain design steps, for instance by mapping high level design phases onto the right low level design operations. Also, designers may be advised about the applicable design operations based on a particular end-goal and based on earlier design decisions. Yet another example of design automation is that from the information registered in a design flow and a certain design state, the design flow system can infer a number of intermediate design operations and execute them automatically. The automation of part of the design process may relieve designers of straightforward, repeating design operations. This may further speed up the design process.

We started this section with the observation that, in order to improve design productivity in complex design processes, design systems should become smarter. We think design flow management can play an important role in the development of smart design systems. Due to its ability to perform functions such as design tracking, constraint enforcement, design assistance and design automation, a design system with design flow management helps designers to perform their design tasks in a correct and efficient way.

1.3 Initial Approach and Requirements

Now that we have explained the essence of design flow management, we can more precisely formulate the approach we take in this thesis to the development of a design flow system. Generally speaking, the development of a design flow system takes two major efforts:

- We have to develop a *design flow model* offering constructs to describe the specifics of a design process in a design flow.
- We have to work out the four general functions of design flow management mentioned above to arrive at a set of more detailed *design flow management functions* that do indeed assist designers in efficiently executing their design tasks.

When developing such a design flow model and such design flow management functions we must have a clear idea about the requirements that the resulting design flow system should satisfy. In this thesis we observe the following global requirements:

- We like the design flow system to be usable for a *wide range of* differently operating design *tools*. If a design flow system does only support part of the available design tools it will never be capable to really assist designers.
- We strive for a design flow system that assists the designers in a thorough way. For instance, we like the design flow system to track the design process in such detail that the tracked design state does indeed help designers deciding what to do next. We call this *fine-grain design flow management.*
- We do not like the design flow system to force designers and tool developers to change their operating procedures. They should be able to reap the fruits of design flow management without offering the way they perform their design activities. We catch this characteristic under the name *unobtrusiveness.*
- We like the *setup time* of the design flow system to be as *small* as possible. After all, if it takes too long to configure such a system, this will have a negative impact on the productivity improvements to be gained.

Throughout this thesis we use these requirements to weight the pros and cons of the different alternatives. Whenever appropriate we will refer to these requirements explicitly.

1.4 Origination of Our Design Flow Paradigm

The development of principles for design flow management in Delft has been an effort spread over a number of people and a number of years. Our initial starting point was the existing, but ever evolving Nelsis CAD framework, a software infrastructure developed at Delft University of Technology, which has been used for testing many new ideas in CAD framework technology. Our goal was to extend this system with design flow management capabilities. The first prototype (1991) contained a large number of Nelsis-specific design flow constructs. Descriptions of the capabilities of tools were expressed directly in terms of Nelsis data management calls. The system did not have any constructs to describe the structure of tool invocation command lines, nor did it have any mechanisms for design automation. However, it was a good starting point for further exploration. The prototype system has been tried and evaluated in various design environments, with completely different tool sets. From these evaluations, new requirements evolved, which were usually focussed at making the design flow system more flexible and easy to configure. These experiences motivated us to extend our design flow system with new constructs and to widen its capabilities. In this thesis we try to step away from the Nelsis-specific concepts (at least in chapters 1 through 5), and to describe what we have found to be generic principles

for design flow management. It must be noted however that these principles could not have been developed without the experiences gained by building prototypes in Nelsis.

1.5 Overview of this Thesis

As described above, the challenge we face in this thesis is to develop concepts for design flow management that can be used to better assist designers during the design process. Roughly sketched, we must develop a model to describe a design process in a design flow (a design flow model) and we must develop a number of functions to assist designers during the design process based on the information recorded in such a design flow. Before going into the modeling of design flows or the development of design flow management functions, in chapter 2 we first give a global overview of design flow management. We explain the basics of CAD frameworks and summarize a number of important design flow management approaches. We present a global architecture of a smart design system and lay the foundations for the creation of a general design flow model. For the development of a design flow model we take a bottom-up approach: we start by observing the characteristics of the design tools and developing a model that fits to the reality of the electronic design process rather than starting from an idealized model of a design process and trying to fit the design tools into this model. Therefore, in chapter 3 we start the development of a design flow model by analyzing the behavior of CAD tools in a framework based design system. This results in an information model that offers a number of constructs to describe tool characteristics important for design flow management. The subject of chapter 4 is design flow modeling. An important aspect of the design process to be described in a design flow is the set of dependencies between the design tools. Together with the development of a design flow model we describe some of the design flow management functions, such as constraint enforcement. Chapter 5 focuses on the interaction with the end-user. It explains how to present the state of the design process in the context of a design flow and how the design flow system can interact with the designers based on such a flow-based representation of the design state. In chapter 6 we focus on the implementation of the design flow system into the Nelsis CAD framework. We show how the concepts presented can be utilized to extend the Nelsis CAD framework with design flow management and investigate how to let the design flow mechanisms closely cooperate with other CAD framework services. In addition, we give a number of examples of how the design flow system has been used in different design applications. In chapter 7 we present our conclusions.

2

Design Flow Management

2.1 Introduction

In this chapter we present a global overview of design flow management. Since we defined design flow management as a set of CAD framework functions (as stated in chapter 1), we start with a brief introduction to CAD frameworks. We spend some words on the history of CAD frameworks, we describe the different CAD framework services and we position a CAD framework in the global architecture of a design system. After this, we focus on design flow management issues. To get a feeling of the state of the art in design flow management, we take a look at a number of prominent approaches in this area and we come to a number of global observations about design flow management.

In the second part of this chapter we present our global approach to design flow management. We define a number of basic concepts, we define the terminology to be used and we define the global architecture of the design flow system by identifying and positioning its major components. In addition, we highlight a number of aspects of design flow management that we have found to be essential for building a design flow system that actually assists designers in a real-world design environment. For each of these aspects we consider the effects of choosing different alternatives on the functionality of the resulting design flow system. At the end of this chapter, we examine in more detail how to describe the characteristics of a design process in a design flow. We explain the notion of a *design flow model* and we make a first-order subdivision of such a model based on the different types of information to be distinguished in a design flow.

2.2 CAD Frameworks

The acceptance of CAD frameworks in the field of electronic design has been a gradual process. In the CAD-prehistory, when only a few design tasks could be automated, there was no notion of a CAD framework. Tools read their input and produced their output from and to files. Because many tools used their own (vendor-specific) data format for storing design information, the interaction between the design tools was a major bottleneck. The first attempt to let the individual design steps more easily communicate was the definition of a number of standards for the representation of design data, like the Caltech Intermediate Form (CIF) and the Electronic Design Interchange Format (EDIF) [EDIF87]. The fact that many design tools internally still used their own data format, which implied a lot of transformations between the data formats, does not diminish the value that design tools could at least read each others data.

The next step that led to a further integration of design tools into one integrated design environment was the use of a central data repository for the storage of design data. This made it possible to offer a uniform way of accessing design data to the design tools and to prevent conflicting accesses to design data. Especially in design areas where large teams of designers work in a multi-user, multi-tasking computing environment, the control of concurrent accesses to design data, indicated by *concurrency control*, has been a major topic. With the introduction of a central data repository and facilities that control the access to the data, the first CAD frameworks were born.

The CAD Framework Initiative (CFI), a public organization that has the objective to define standards for CAD, defines a CAD framework as [CFIugo90]:

Definition 2.1:

A CAD framework is a software infrastructure that provides a common operating environment for CAD tools.

Since, with the acceptance of a central data repository, all design tools access their design data via a CAD framework, the services offered by a CAD framework to the design tools and to the designers could be extended with more powerful *data management* services. We explain a number of these services in more detail:

1. *versioning*:
   Since designing is an iterative and tentative process, in which designers typically try different design solutions before a satisfying one is found, most CAD frameworks support the existence of a number of *versions* or *alternatives* of one and the same design. This enables designers to experiment with different implementations before choosing one of them. To

keep track of how different versions of a design relate, the derivation relationships between these versions are administered as well. This results in a *version graph*, which, in principle, is a directed acyclic graph.

2. *viewtypes*:
   In complex design applications, it is quite common to have multiple representations of a single design. For instance, in electronic design a specific component may have a description at the behavioral description level, at the register-transfer level, at the gate level and at the layout level. CAD frameworks that support this data organization allow the definition of a number of *viewtypes*, which correspond to the possible representations of a design.

3. *hierarchy*:
   In many design applications it is convenient to decompose a design into smaller parts, which on their turn can be decomposed into smaller parts, etc. The decomposition of a design into *subcells* can be described by a set of parent-child relationships. Each parent-child relationship denotes an *instantiation* of a *child* cell into a *parent* cell. The directed acyclic graph consisting of edges for parent-child relationships and nodes for cells is referred to as a *design hierarchy*. The advantage of a hierarchical decomposition is that parts that occur more than once in a design can be described once and instantiated many times. Apart from a considerable reduction in data storage for large designs, the decomposition of a design into subcells may also benefit the applicability of *concurrent design* since different design teams may be allocated to different subcells.

4. *access control*:
   In complex design environments it may be useful to distinguish between users with different *access permissions*. An access control mechanism enables the assignment of access permissions to users and checks the design operations performed against the configured access permissions. Depending on the granularity of the access control mechanism it may hand out permissions to perform general operations, like the permission to create, alter or delete design data, the permission to define new viewtypes and users, the permission to approve a design etc, or it may distribute very detailed permissions, like the permission to work on a specific design cell. Since in many design applications the permissions of a user depend on the *role* he plays in the design process (such as designer, tool developer, design manager etc.), most access control mechanisms allow roles to be assigned to users and permissions to roles. In this case the permissions assigned to a role apply to all users with this specific role.

In addition to data management facilities, CAD frameworks typically offer a common user interface from which all design operations can be invoked and which presents the state of the design to the designers.

Although many quite different approaches to CAD frameworks exist, there is at least some agreement on the global architecture of a CAD framework. A global CAD framework architecture as accepted by CFI [CFIfar93], the Jessi Common Framework project (JCF) [JCFarch91] and Nelsis [Wolf94a] is shown in figure 2.1.

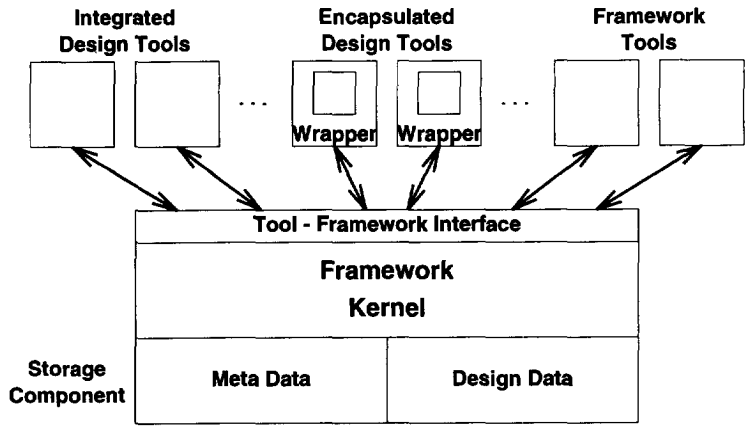


Figure 2.1. Global architecture of a CAD framework.

It shows multiple tools communicating with a CAD framework via the *tool-framework interface*. These tools may be operated by different users and possibly from different machines. Tools perform requests to the CAD framework to get access to the design data stored in the *storage component*. The storage component is divided into two parts, one for the *design data* and one for the *meta data*, which contains information registered by the CAD framework *about* the design data and the design process. Design data is usually organized in *design objects*. A design object is the smallest unit of design data on which meta data administration is performed. Administrative information, such as versions, viewtypes and hierarchies are stored in the meta data with references to the corresponding parts of the design data. This data organization benefits to the efficiency of the CAD framework, since for collecting data about the design state it does not have to access the usually larger collection of design data but can restrict itself to the usually smaller and efficiently organized meta data.

The *framework kernel* is that part of the software infrastructure with which the tools communicate at run-time. This part of the CAD framework implements all framework services that have to be performed centrally, for instance because of reasons of data consistency. All communication of the CAD framework with the end-user is performed through *framework tools*. An example of a framework tool is a hierarchy browser, which presents the hierarchical decomposition of (part of) a design in a graphical way.

Design tools can be divided into *integrated* and *encapsulated* design tools. Integrated design tools have CAD framework requests placed directly into their source code. This is the preferred method of tool integration for complex design tools that need to interact with the CAD framework at run-time. However, the modification of tool source code is not always possible or desirable. For instance, if so called *third-party* tools must be integrated in a design environment, it is usually impossible to add CAD framework requests to their source code. In these cases tools can be incorporated into a CAD framework by writing a piece of code, called a *wrapper*, which handles the communication with the CAD framework. Some frameworks offer a special *extension language* for this purpose, which usually contains a number of easy to use, high level programming constructs that give access to the tool-framework interface.

Examples of commercial CAD frameworks are Design Framework II (Cadence), the Falcon Framework (Mentor), Powerframe (DEC), OpenFrame (Viewlogic), SiFrame (SNI) and ProFrame (IBM). A number of CAD frameworks have been developed for research purposes. Some of the more well-known are Oct/VEM [Harrison86], Pace [Sarmento91], Star [Wagner92], Odyssey [Brockman92] and Nelsis [Wolf94a].

2.3 Design Flow Management Approaches

Although not all data management problems have been solved completely, research in CAD frameworks has been moving to other topics. A subject that attracts a lot of attention considering the large number of publications in this area, is how to extend a CAD framework with facilities to manage the design process rather than just managing the design data. Research in this area is still ongoing. Although CFI tries to develop standards in this field [Fiduk90, CFItes92], no generally accepted approach has been identified yet. There is even no commitment on the terminology to be used for these kind of services. Some of the more popular terms being used are *design flow management* [Bretschnei90, Bosch91, Barnes92, Bingley92], *task management* [Brockman91, Chiueh93], *design methodology management* [Fiduk90], *design process management* [Hamer90, Jacome92, ProFrame94], *process scheduling* [Tanaka90] and *design decision support* [Beggs92]. As described in chapter 1, we have chosen to use the term *design flow management* in this thesis.

An extensive analysis of a number of different approaches to design flow management can be found in [Kleinfeldt94] (called design methodology management in this publication). The systems reviewed are classified according to a number of criteria, such as "level of abstraction", "invocation definition", "flow definition" etc. Although this classification gives an excellent overview of the state of the art in the field of design flow management, it also makes clear that no commonly accepted approach exists yet and that research is still needed to "effectuate the productivity improvements that design flow systems promise".

Below, we describe a number of prominent approaches to design flow management. The order in which they are described is chronological (based on their first publication).

2.3.1 Monitor

Di Janni [Janni86] has been one of the pioneers in the area of design flow management. Using his "Monitor for complex CAD systems" a design environment of tools operating on files can be configured in an extended Petri net. A graphical user interface presents the state of the tools and files using traffic light colors as a metaphor. This approach has been important because it showed that a powerful user interface for designers can be based on a formal description of the design environment.

2.3.2 VOV

The VOV system [Casotto90], developed at Berkeley University of California, takes a quite different approach. Instead of configuring a design flow, tools are instructed to leave a trace of their execution. The set of tool execution traces forms a graph called a *design trace*, which contains the dependencies between data and tools. If data becomes out of date, a *retracing* process invalidates old data and tools are re-executed automatically. It is possible to run the system in a mode in which it uses a design trace as a "program" that defines a design flow. This system has attracted a lot of attention for its unusual approach to design flow management. It was the first to apply a tracing approach in the field of design flow management.

2.3.3 The CAD Framework Initiative

The CAD Framework Initiative (CFI) delivered several documents concerning design flow management. [Fiduk90] gives a high level overview of the work of the Design Methodology Management Subcommittee (DMMTSC). Its main results are the decomposition of a design flow system into three parts: tool encapsulation, task flows and execution environment. Especially in the field of tool encapsulation CFI has been active. This resulted in a tool encapsulation specification (TES) [CFItes92], which describes a human and computer readable language for encoding encapsulation information of a design tool. Although this specification contains constructs for describing data access characteristics, environment variables etc., its main power is in the specification of the syntax of the command line of a tool.

2.3.4 MMS

The MCC CAD Framework Methodology Management System (MMS) [Allen90, Allen91] focuses on design flow (methodology) management in the field of distributed engineering. It addresses load balancing in a distributed computing environment and the distribution of design activities among designers. A methodology description consists of tool and task descriptions, expressed in an extension language. Pre- and post-conditions can be specified for tasks. Tasks can be associated with specific users. Tool and task descriptions can be linked into an executable 'methodology' which is a directed graph with nodes for tools and arcs for dependencies between tools. The type of dependency is defined in an extensible dependency class hierarchy.

2.3.5 Hilda

The HILDA [Bretschnei90, Bretschnei92] system offers a "knowledge based design flow management" approach. Design flow management has been implemented using a mixture of Petri nets and production rules with certainty factors. Petri-nets were chosen for modeling the design flow, because of their visual and graphical quality. More complex design flow knowledge like decision support and parameter selection is described in production rules. Internally, both nets and rules are transformed into production rules, which are fed into a knowledge base with inference engine. This approach is remarkable for combining Petri-nets and a knowledge base. Apparently the authors felt that one formalism could not provide enough modeling power.

2.3.6 Roadmap

The Roadmap system [Hamer90], developed at Philips Research Laboratories, has been important for three reasons:

- It applies semantic data modeling in the field of design flow management.
- It uses a data flow model for describing design flows.
- It combines the usually separated issues of design process management and design data management into a single flow paradigm.

The backbone of this system is a three layer information architecture for describing the available design tools (layer 1), the data flow between design tools (layer 2) and the dynamic aspects of the design environment (layer 3). For systems based on this information architecture, a browsing technique is described via which designers can inspect the interrelationships between different parts of their design. Tools that accept their own output data as input, like editors, require an additional construct for optional data usage.

2.3.7 Nelsis

The design flow management module of the Nelsis CAD framework [Bosch91, Bingley92, Bosch93, Wolf94b, Bosch94] takes a data flow based approach to design flow management. It supports a wide range of tools, varying from interactive tools to batch tools. The information concerned with design flow management is divided into configuration information, which contains a description of the static properties of a design environment, and run-time information, which administers the state of the design process. Central in this approach is the notion of an *activity*, which corresponds to a design function of a

design tool. Design flow management is performed at the level of activities rather than at the level of tools. Special attention is given to the problem of integrating design flow management into a CAD framework without altering the tool-framework interface, so that existing design tools do not need to be modified. A user interface is described that presents relevant parts of the design state to the designers.

An extensive discussion about the differences and similarities of the Roadmap approach and the Nelsis approach can be found in [Hamer91].

2.3.8 Odyssey

The Odyssey CAD framework [Brockman92], developed at Carnegie Mellon University, contains a module for task management, Hercules, and a module for design process management, Minerva. The Hercules Task Management system [Brockman91, Sutton93], uses a *task schema* to define abstract design functions and their relationships. Designers can extract a subgraph of the task schema, called a *task tree*, which is a local data structure needed to execute a task. Special about this approach is that it can handle tools that were created during the design process. The design process manager, Minerva [Jacome92, Lopez92], offers a mechanism for planning and managing the design process. It exploits a hierarchical definition of design domains to assist designers in making the right design decisions.

2.3.9 The Jessi Common Framework Project

The Jessi Common Framework (JCF) approach to design flow management is described in [Liebisch92]. This approach offers extensive support for teams, user roles and workspaces. A flow defines the relationships between the activities in a design project. This approach differs from others in the sense that it allows flows to be specified *per cell*. Therefore the dependencies in a flow can be restricted to temporal dependencies.

2.3.10 DAMOCLES

The DAMOCLES design tracking system [Vasudevan92, Mathys93] developed by Motorola, has an observer-based architecture. Instead of replacing existing operating environments it augments operating environments by observing the design process. It administers a project-wide design state, which can be inspected by designers using queries. Users can define validation policies, which are used to validate or invalidate designs if certain database events occur (validation management). The design tracking approach of DAMOCLES has some

similarities with the design tracing approach of VOV (see above). Together they represent an interesting branch of the design flow management research community.

2.3.11 Concluding Remarks

From this overview we make the following global observations:

1. There is a clear distinction between design flow management approaches that assist designers based on a *traced* description of the design process (tracing-based approaches) and approaches that assist designers based on a *configured* description of the design process (configuration-based approaches). Although the configuration-based approach is more popular, we cannot simply conclude that this approach is better.

2. Most of the approaches use some kind of graphical 'design flow' representation to display the state of the design process to the designers and to further assist them during their work. Apparently, the use of such a representation is suited for this purpose. However, there is a large variety of representations in use. Most approaches use some more or less formal model for generating these representations, but the number of different models used for this purpose is large. Petri-nets definitely belong to the more popular ones.

3. Some of the approaches recognize a number of different layers in their model for describing a design process. For instance, CFI discriminates between tool encapsulations, tasks flows and execution environments, MMS distinguishes between tool and task descriptions, and Roadmap makes a distinction between design tools, data flows between design tools and dynamic aspects of the design environment.

These observations address a number of important aspects of design flow management and design flow models, which need some further discussion. We let this discussion concur with the presentation of our global approach to design flow management. In the next section we present our view on these aspects of design flow management and at the same time we refine the initial approach presented in chapter 1. We discuss observations 1, 2 and 3 in more detail in sections 2.4.3, 2.4.5 and 2.4.6 respectively.

2.4 Global Approach

In this section we address a number of essential issues that came out of the overview in the previous section or that we have found to play a prominent role in design flow management research. We start with a more detailed examination of the smartness that we ascribed to a design flow system. This results in two types of information to be distinguished in a CAD framework with design flow management. After that, we examine the global architecture of a design flow system in relation to the general architecture of a CAD framework as presented in section 2.2. Then, we explain our view on the issue of tracing-based design flow management versus configuration-based design flow management, which is one of the observations of the previous section. Finally, we focus more specifically on the structure of a design flow and we describe the strategy that we follow in the subsequent chapters for modeling a design process.

2.4.1 From Administration to Configuration

In chapter 1 we motivated the development of a design flow system with the observation that design systems should become smarter. Of course, design systems that do not offer any design flow management functionality but that do offer facilities for data management already offer some kind of smartness to the end-users. However, this smartness is essentially different from the smartness we expect from a design flow system. The difference can be explained from the type of information that plays a role in both types of design systems.

The information traditionally recorded by most CAD frameworks describes the *state* of the design process, such as the available design data, the dependencies among the different design data entities and the operations performed on this data. The addition of design flow management to CAD frameworks introduces a totally different information need. In order to better assist designers in performing their complex design tasks, a design flow system must also have at its disposal information about the *structure* of the design process, such as the available design tools and their interdependencies. It is this kind of information that is the key to the smartness of a design system with design flow management.

A design flow system can be configured for a specific design application by feeding it with knowledge about the structure of the design process. This makes that in addition to an extensive *administration* of the state of the design process, a design flow system needs to have at its disposal the notion of a *configuration*, which describes the structure of the design process. It is useful to make a distinction between the information that has to do with administering the design process and the information that has to do with configuring the design process.

Naming the two types of information after the point in time that it is generated (at run-time or at configuration-time), we come to the following subdivision:

- *run-time information.*
  This is information that is generated during the design process to administer the state of design. Run-time information is *dynamic* in the sense that it is updated continually in the course of the design process. Examples of run-time information are the different versions of a design or the operations performed on a certain design.
- *configuration information.*
  This kind of information is usually defined before the design process starts. Configuration information is static in the sense that it typically changes only if the structure of the design process is changed in some way.

As described in chapter 1, a design flow system assists designers based on a design flow. For this reason, we call the information that configures a design flow system for a specific design application the *design flow configuration.* Generally speaking, the design flow configuration contains a description of the structure of the design process in terms of the relationships between integrated and encapsulated design tools and framework tools. Figure 2.2 gives a graphical impression of the notion of a design flow configuration.

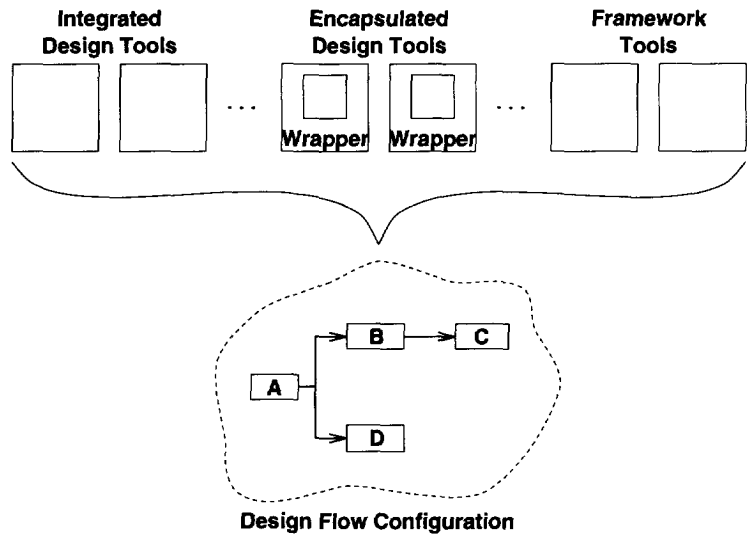


Figure 2.2. The design flow configuration describes the design process in terms of the tools and their interdependencies.

2.4.2 Global Architecture of a Design Flow System

As stated in chapter 1 we see design flow management as one of the services of a CAD framework. To keep the concepts presented in this thesis generally applicable we make minimal demands on the architecture of such a CAD framework. The first assumption we make is that tools communicate with the CAD framework through a tool-framework interface that offers at least functions to access the design data stored in the storage component that is under the control of the CAD framework. This assumption is necessary to allow a design flow system to perform functions such as design tracking and constraint enforcement. Second, we require that the CAD framework consists of a number of software modules with well-defined interfaces. This guarantees that we can add design flow management, without having to redesign the other parts of the CAD framework. We feel that these assumptions are so general that they do not restrict the applicability of the design flow concepts. Any well-designed CAD framework that conforms to the generally accepted CAD framework architecture of figure 2.1 satisfies these assumptions. In this section we examine in more detail how a design flow system fits into this architecture.

In the general CAD framework architecture of figure 2.1, CAD framework functionality is spread over a framework kernel and a number of framework tools. Because of the designer assisting character of design flow management, it is inevitable that a design flow system offers means to communicate with the designers. For instance, it must inform designers about the state and history of the design and advise them about prospective design steps. According to the general framework architecture presented above, the communication with the end-users should be performed by framework tools. Therefore, without saying anything about the specific functionality of the different components, we may conclude that the design flow system should at least contain a *design flow user interface*, implemented as a framework tool.

An essential question is whether the entire design flow system could be implemented as a framework tool, or whether at least some functionality should be put in the framework kernel. An important observation with respect to this question is that if the entire design flow system is implemented as a framework tool, it cannot track the communication of the other tools with the CAD framework. This limits the assisting power of a design flow system in two ways:

- Although the design flow system does know about the beginning and the termination of tool runs invoked via its user interface, it cannot track the detailed operation of design tools within the scope of a single tool run. This means that the operations performed via tools that decide at run-time which

function to perform (such as interactive tools) cannot be tracked (and displayed) until such tools terminate.

- The design flow system can only track the execution of design tools that were explicitly started via the design flow user interface. This has some serious consequences. First of all, design operations performed via design tools that were not started via the design flow user interface are not recorded in the state of design administered by the design flow system. This results in an incomplete (or wrong) representation of the state of design to designers. Second, these design operations are not checked against the configured design constraints. Hence, it cannot be guaranteed that all design operations conform to a configured design flow.

These limitations make that in an architecture where design flow management is implemented as a framework tool, the design state cannot be tracked on a finer granularity than in terms of completed tool runs, and that in order to let the design flow system properly track the design process and enforce design constraints, designers are forced to use the design flow user interface to invoke their design tools.

If part of the design flow system is implemented in the kernel of the CAD framework, the above limitations do not longer apply. Since the tools communicate via the tool-framework interface to access their design data, this part of the design flow system can track the detailed operation of all design tools while they are still running and it can enforce all design constraints configured in the design flow configuration. This holds for design tools that were not invoked via the design flow user interface as well.

Since in this thesis we want to develop a generally applicable design flow system that offers detailed assistance to designers (fine-grain design flow management) and that does not force designers to perform their design operations in one specific way (unobtrusiveness), we choose to integrate at least the design tracking and the constraint enforcement functionality of the design flow system into the framework kernel. We name this part the *design flow kernel*.

This brings us to a global architecture of the design flow system where the design flow facilities are spread over a *design flow kernel*, integrated in the framework kernel and a *design flow user interface*, implemented as a framework tool. Both parts must have access to information about the structure of the design process described in the design flow configuration. Figure 2.3 shows the architecture of a CAD framework based design system with design flow management. The design flow components are shaded.

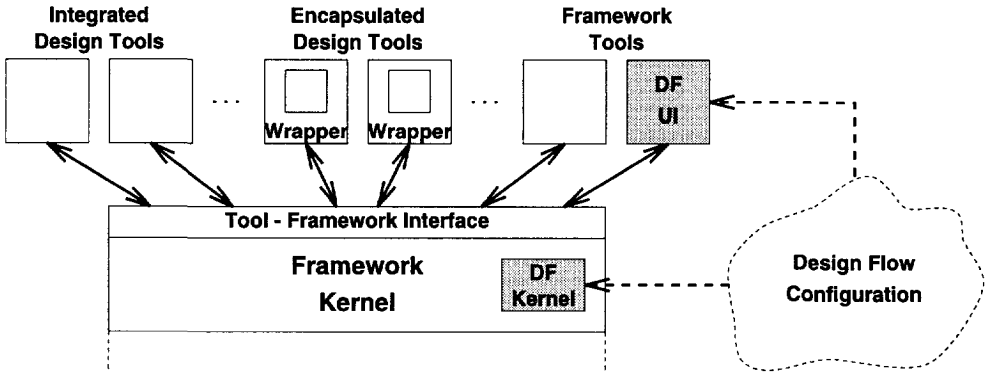


Figure 2.3. Architecture of a CAD framework based design system with design flow management.

2.4.3 Configuration-Based versus Tracing-Based Design Flow Management

From the overview of the design flow approaches in section 2.3 we observed a fundamental difference in the way the different systems handle a design flow. At one side of the spectrum we have the *configuration-based* design flow management systems [Janni86, Bretschnei90, Hamer90, Brockman91, Bosch91]. These systems are characterized by the existence of a predefined design flow to which all design operations must adhere. Design operations must be configured before they can be used. At the other side of the spectrum we have the approaches taken by VOV [Casotto90] and DAMOCLES [Vasudevan92, Mathys93]. In these systems there is no notion of a predefined design flow. Designing can start without first configuring a design flow. During the design process the system collects information about the design operations performed and uses this to derive a (graphical) representation of the design process. We term this approach *tracing-based* design flow management.

A major advantage of the tracing-based approach is that it takes no configuration time before the design process can be started. This is a desired feature with respect to the requirement that the setup-time of the system should be as small as possible. However, a disadvantage is that the resulting system can only be used to *track* the design process. It cannot be used to *enforce* design constraints configured in a design flow. Another difference between the two approaches is that in the configuration-based approach user interaction can be based on a configured description of the design process, where tracing-based systems do not have such a description. We discuss this issue in more detail in chapter 5.

In addition to pure configuration based and pure tracing based systems, a number of approaches have been described that offer a mixture of both. In [Casotto90] an additional mode is described that uses a recorded design trace as a design flow definition and in [Bingley92] a configuration based design flow system is proposed, which also offers a learning facility to operate in tracing mode.

As specified in chapter 1, the design flow system must be able to enforce the design constraints configured in a design flow. This implies that a pure design tracing approach is not suitable as the basic mechanism in our design flow system. On the other hand, the need to speed up the overall design process pressurizes the time available to configure a design system. This makes a tracing-based approach attractive because of its minimal design flow setup-time. In fact, the need for design flow enforcement and a short setup time are competing requirements. We believe that the ideal design flow system offers multiple modes of operation and allows to switch from one to another. This makes it possible to use different modes of operation in different stages of the design process. In a preliminary stage of the design process the tracing mode could be used to automatically enrich the design flow system with knowledge of the design process and in a later phase the information collected could be utilized to assist designers by running the system in a checking mode.

Summarizing, we come to the following modes of operation:

- *checking*:
  The design flow kernel monitors the design operations performed by the design tools and checks them against the configured design flow.

- *tracing*:
  The design flow kernel traces the design operations performed by (a subset of) the design tools and stores this trace in a human and computer readable form. This trace can be used for analyzing the operation of a tool. When running the design flow kernel in trace mode, it does not prohibit design operations that do not conform to the (partly) configured design flow.

- *learning*:
  The design flow kernel traces the design operations performed by (a subset of) the design tools and matches them against the configured design flow. If the match fails, the design flow kernel tries to adapt the design flow definition to the observed tool behavior.

Figure 2.4 shows the three modes of operation of the design flow kernel.

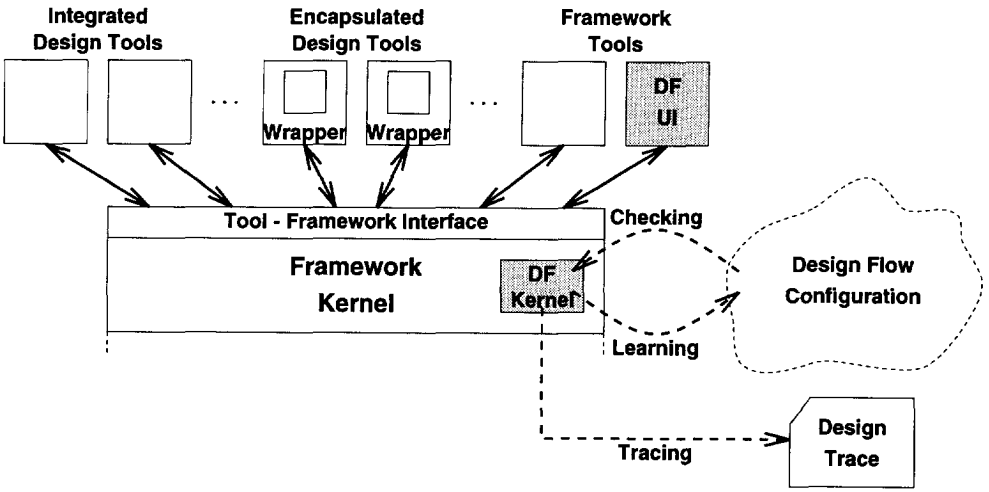


Figure 2.4. Three modes of operation of the design flow kernel.

In this thesis we do not describe the learn mode and the trace mode in detail. We concentrate on what we believe should be the 'default' mode of operation of a design flow system: the checking mode. We feel that the learn and trace modes are particularly useful to facilitate the definition of a design flow configuration, which we describe in more detail in the next section.

2.4.4 Design Flow Acquisition

From the previous sections it follows that the key to the smartness of a design system with design flow management is located in the design flow configuration. In this section we examine in more detail how a design flow configuration can be created and refined. We capture this under the name *design flow acquisition*. The learn mode mentioned before offers only one of the three methods for design flow acquisition. A second method is to exploit the information recorded in a design trace to update the design flow configuration and a third method is the manual definition of a design flow configuration. This brings us to the following three methods for design flow acquisition (see figure 2.5):

- *manual*:
  A *flow configurator*, someone with enough knowledge of the tool set and the design process, configures the design flow manually. This may demand a considerable effort of the flow configurator. To make manual configuration easier (which is in line with the requirement for a small setup-time), the design flow system may offer a dedicated *design flow editor*.

- *semi-automatic*:
  By running the design flow system in trace mode while performing a number of characteristic design activities, a (maybe incomplete) description of the design process can be obtained. The resulting design flow trace can be inspected by the flow configurator to decide whether or not the design flow should be updated.

- *automatic*:
  This method is used if the system runs in learn mode. Instead of just recording the recognized design operations, the design flow kernel tries to adapt the configured design flow to the design operations performed. This way of design flow acquisition is particularly useful in simple design environments that require little advanced design knowledge. An experienced designer can simply demonstrate the design strategy to the system, which is recorded and added to the design flow definition (replay facility).

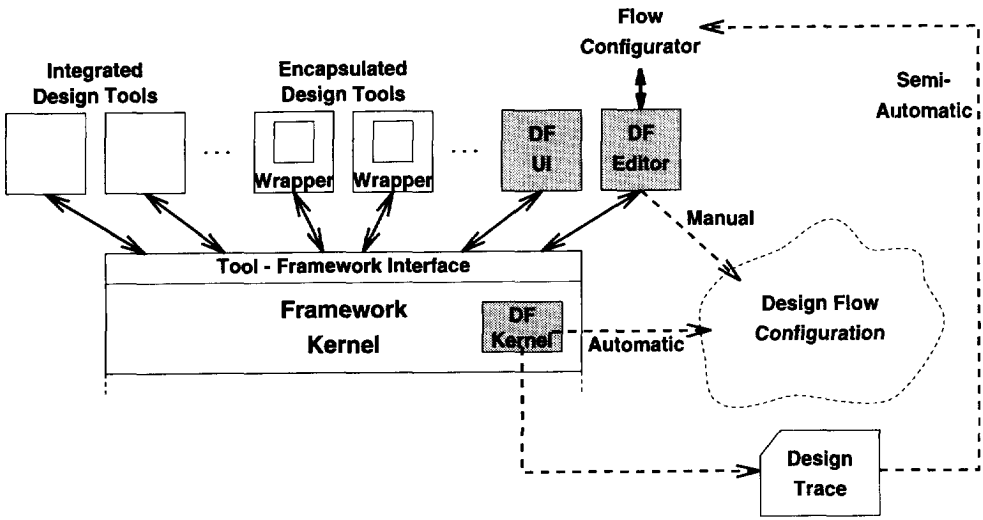


Figure 2.5. Three methods of design flow acquisition.

In our view, the degree in which design flow acquisition can be automated depends on the complexity of a particular design process and the level of detail that needs to be used when describing the design process. In simple design environments where the design flow simply follows from the available tools, automated design acquisition may yield a satisfying result. In complicated design environments however, where a lot of specialized design experience must be brought in, the involvement of a human being will be inevitable. We think that no matter how clever the design flow kernel is constructed, the system must offer a

way to take the specialized knowledge of the flow configurator into account during design flow acquisition. On the other hand, tracing and learning facilities may drastically cut down the design flow acquisition time, which is a desired feature with respect to the need to speed up the overall design process. Therefore, we take the position that a proper design flow system should offer multiple methods of design flow acquisition varying from manual to fully automated.

2.4.5 Modeling a Design Process

In the previous sections we made no assumptions on the structure of a design flow configuration yet. We just stated that it contains a description of a design process to be used by the design flow system to assist designers during the design process. However, to let the design flow system perform the design flow management functions described in chapter 1, it is essential that the information recorded in a design flow configuration is well-structured and tuned towards its use in a design flow system. Therefore, we must describe exactly which constructs are available to describe a design process and what their meaning is. This information is summarized in a *design flow model.*

Definition 2.2:

A *design flow model* is a collection of constructs to describe a design process in a design flow.

The task of finding an appropriate design flow model to describe the structure of a design process appears to be one of the most difficult problems in design flow management research. Actually, one of the observations we made as a result of the overview of different approaches to design flow management (in section 2.3), was that there are many different models in use for this purpose. Some of the approaches use Petri nets (for instance, colored Petri nets and predicate transition nets), others use so-called data flow graphs. Although we subscribe the use of a formal model to describe a design process, we have the feeling that some of the models used are actually not very well suited to describe a complex design process. In many cases the model chosen has to be extended with additional constructs to grab the reality of the design process.

As an example, consider the HILDA Petri net approach discussed extensively in [Bretschnei92]. The initially chosen predicate transition nets have to be extended with arc-types for non-consuming access, arc types for optional inputs, special control places and tokens to prevent tokens on outputs from firing a transition, arc types for multiple inputs and outputs, and hierarchical places for a hierarchical definition of the net. In our view, these extensions make it questionable whether a predicate transition net is actually a natural way to describe a design process.

To avoid a premature choice for one specific model, in this thesis we take the position that a detailed model for describing design processes cannot be developed without carefully inspecting the characteristics of design processes in general. To provide a design flow system with useful knowledge of a design process, we have to know which characteristics to describe and how. Only then, we can develop constructs to describe a design process in a design flow. Therefore, in this thesis we first carefully analyze which characteristics of a design process should be described in a design flow, then we develop for each characteristic one or more appropriate design flow constructs and finally we describe the relationships between these constructs in an *information model*. This information model formally describes the available design flow constructs and their relationships.

To describe such an information model we use the OTO-D (Object Type Oriented Data) modeling technique [Bekke92]. Since the use of this modeling technique in the area of CAD frameworks has already been discussed extensively in various other publications (see for instance [Wolf88] and [Wolf94a]), we do not further motivate this choice here. For a brief introduction to OTO-D see appendix A.

Above, we mentioned that there are many different models in use for describing design processes. In this thesis we aim at a design flow model which is optimized to the following general characteristics:

- *expressiveness*:
  In order to assist designers in a detailed way (fine-grain design flow management) a design flow system should have at its disposal detailed knowledge about the design process. Therefore, we aim at a design flow model that contains a number of *expressive* constructs to describe the reality of a design process in a design flow.

- *flexibility*:
  It is a common situation that some characteristics of a design process may not be known at design flow configuration time. For instance, the exact operation of interactive design tools can usually not be described at design flow configuration time. For such characteristics it is important that the design flow model provides *flexible* constructs so that characteristics can be described in detail if they are known and more vaguely in other situations.

- *generality*:
  If a design flow model is limited in the type of design processes it can describe, this also limits the applicability of the design flow system as a whole. Therefore, the design flow model should have constructs that are *general* enough to describe any type of design process.

2.4.6 Tool Modeling versus Design Flow Modeling

In this section we view the contents and structure of the design flow configuration in more detail. In section 2.3 we already observed that many approaches to design flow management recognize one or more layers in their model for describing design processes. We agree with these approaches that a proper subdivision of the design flow configuration may benefit the task of design flow acquisition and the use of the design flow system as a whole. However, a necessary condition for this is that the subdivision is based on a meaningful difference in the use of the information in a design flow configuration.

One such meaningful subdivision is based on the difference between information that describes the capabilities and characteristics of design tools and information that describes the use of design tools in a certain design process. We call the information that describes the characteristics of a single design tool a *tool description*. A tool description contains information about the capabilities of a design tool, without taking the context in which the tool is used into consideration. Ideally, tool developers should accompany their CAD tools with a tool description that can be inspected by a design flow system to get familiar with the specifics of the design tool in question. Such a tool description should describe the characteristics of a design tool that are relevant for design flow management. Examples of such tool characteristics are:

- the purpose of a tool,
- how to invoke a design tool,
- the data requirements of a tool,
- the expected data production of a tool,
- whether or not user interaction is required,
- a description of the environment variables the tool inspects.

The relevance of a tool characteristic depends on whether the design flow system can use it to assist the designer. For instance, the fact that a design rule checker always reads a design rule specification that is stored centrally, out of the designers control, is of no interest to the designer. On the other hand, the fact that a design rule checker generates a design rule error file, if the check fails, and no error file if the check passes without errors, is important to take into account. This means that a design flow system can draw conclusions on the result of the design rule check without even inspecting the error file.

The second type of information that we recognize in a design flow configuration describes the *structure* of the design process, such as the relationships between the design tools. This part of the design flow configuration corresponds to the notion of a *design flow* as introduced in the first chapter. The reason to differentiate between tool descriptions and a design flow is that a tool may be used in different design processes in which it may play a different role. To avoid the need to duplicate the description of a design tool in each design process in which it is to be used, the tool descriptions can better be separated from the design flow descriptions. In this way, different design flows may be defined for different design processes, based on the same design tool descriptions.

To support the layered structure of a design flow configuration in tool descriptions and a design flow description, we develop a *tool model* independent from the *design flow model*. For each characteristic of the design process we decide whether the design flow system must have knowledge of it, and if so, in which of the models it belongs. For instance, the characteristic of a tool to either read some data entity or to leave it untouched (optional input) is a property of the *tool*, not of the environment in which it is integrated. Therefore, this property should be described in a tool description and not in a design flow. On the other hand, a constraint that tool A must be executed before tool B, is a property of a specific design process. Hence, this property should be described in a design flow.

Some approaches found in literature recognized the benefit of a division of tool modeling and design flow modeling before. In [Cadence90] a seven-layer conceptual model of frameworks is presented. The subdivision into tool management and flow management (among other layers), corresponds roughly to our two layered design flow model. In [Hamer90] a three layer information architecture is presented from which layer 1 and layer 2 have the same purpose as the tool description layer and the design flow description layer in our model. In [Hübel92] a design flow specification is defined according to a design tool model, a design flow model and a design structure model. The design tool model corresponds to our tool description layer and the design flow model to our design flow layer. It is interesting that [Hübel92] also recognizes that the description of the design tools in the design tool model should not depend on the chosen design flow in the design flow model. In [Kleinfeldt94] a distinction is made between a *task* and a *flow*. A task defines how to use a tool and a flow defines in what order to use tools. This corresponds roughly to our tool descriptions and our design flow. Summarizing, the distinction between tool modeling and design flow modeling has been recognized in other approaches as well. We think we may state that this subdivision is generally accepted.

2.4.7 Freedom of Design

An objection that is sometimes raised to the use of a design flow system is that it restricts the designers' freedom of design and is therefore unsuited for certain design applications that require a lot of freedom for experimentation [Rumsey92]. This objection is based on the observation that the specification of a design flow implies a choice for a specific design strategy by predefining the order in which design tools are to be executed and that if the design flow system operates in a checking mode, designers have no way to by-pass the strategy encoded in the design flow. Although this observation is true, we do not share the concerns mentioned above.

It is true that a design flow system can be used to create a very strict design environment, in which designers have little chance to escape from a predefined design strategy. However, this is not the only way a design flow system can be used. First of all, in addition to a strict checking mode a design flow system may also offer a mode in which the design flow configuration is used to generate advices rather than as a set of strict design regulations. Using this mode, designers may profit from the assistance of the design flow system while retaining their freedom of design. Second, even if the design flow system operates in a strict checking mode, it depends on the nature of the design flow model and a specific design flow configuration, in what degree the freedom of the designers is restricted. For instance, if the design flow model allows the definition of multiple alternative design strategies (we call them design alternatives), the design system can be tuned towards maximum design freedom, by configuring all design alternatives that are technically possible, or towards a more restricted design environment, by encoding only the design alternatives that are to be used. In other words, when defining a design flow the design flow configurator may choose his favorite place in the design enforcement spectrum.

Concluding we say that the objection that the use of a design flow system restricts the designers' freedom of design is not generally true. Either the use of a special non-restrictive mode or the definition of multiple design alternatives in a design flow configuration enables the use of a design flow system in strict design processes as well as in very lenient design processes. In this way a design flow system can be used for any type of design application and the designers' freedom of design is guaranteed.

2.5 Conclusion

In this chapter, we laid the foundations for the development of the design flow system in the rest of this thesis. After a general introduction in CAD frameworks and a description of the state of the art in design flow management, we introduced the notion of a design flow configuration as the source of the smartness of a design flow system and we positioned a design flow system in the global architecture of a framework-based CAD system. One of our conclusions is that a design flow system should consist of two distinct modules, a design flow kernel, which handles critical design flow functionality that should be performed centrally, and a design flow user interface, which handles the communication with the end-users.

Another important observation we made is that both configuration-based design flow management as well as tracing-based design flow management have a their pros and cons. In configuration-based approaches it is possible to enforce the design constraints configured in a design flow, but it may take longer to setup the design flow system. Tracing-based approaches on the other hand combine a small setup-time with the inability to enforce design constraints. Our conclusion is that a powerful design flow system should combine the best of both.

An issue that is closely related to the configuring versus tracing dilemma is the process of configuring a design flow, termed design flow acquisition. Although in simple design processes parts of the design flow acquisition process may be automated, we feel that the interference of a design flow configurator is still necessary in situations where the design process is not so simple and where a lot of specialized design experience must be brought in.

At the end of this chapter we focussed more specifically on the structure of the design flow configuration. In this thesis we develop a design flow model that contains a number of expressive, flexible and generally applicable design flow constructs to describe the structure of a design processes in a design flow. To formalize the semantics and the relationships between the different design flow constructs we use information modeling techniques.

With respect to the contents of a design flow configuration, we made a distinction between tool descriptions and design flow descriptions. It is important to model each characteristic of the design process at the right level of description. We use this subdivision to structure the next two chapters. In the next chapter we develop a tool model which offers a number of constructs to describe the characteristics of design tools and in chapter 4 we focus on the description of design flows.

3

Tool Modeling

3.1 Introduction

Design tools are the basic building blocks of any computer aided design process. In the end, all design activities of all designers are performed through the operation of design tools. Therefore, in order to assist designers during the design process, a design flow system must be informed about the capabilities of the design tools available to the designers. As described in section 2.4.6, we suppose for each design tool this kind of information to be encoded in a *tool description*, which is a description of the characteristics of a tool that are relevant for design flow management. In this chapter we develop a *tool model*, which describes the structure and semantics of the information to be contained in a tool description.

Traditionally, information about a design tool is described in the documentation delivered with a tool. For instance, the functionality of a tool, the files it operates on and its command line syntax are usually described in a manual page. The problem with this kind of information is that it is intended for the end-user and not for further use by a design flow system. This means that the information in the tool documentation must be completed and transformed into a format that can be recognized by the design flow system. This task is usually performed by the design flow configurator. It would be much more efficient if tool developers would describe the characteristics of their design tools directly in a standardized and computer readable format. An attempt to formalize and standardize information about design tools in a human *and* machine readable language can be found in the Tool Encapsulation[1] Specification (TES) of CFI [CFItes92].

1. Previous versions of this specification were called Tool *Abstraction* Specification. In our view, this previous name better reflects the initial objective to provide a language for design tool characterization.

Unfortunately, as we will see, this specification turns out to be incomplete for our purpose. It does not cover all tool characteristics that we like to be described in our design flow system. Therefore, for the development of the tool model in this chapter we inspect only those concepts of TES that may contribute to a better description of the characteristics of design tools for the purpose of design flow management. At the end of this chapter we present a more detailed discussion of TES.

For the development of a tool model we have to keep in mind that our objective is not to *define* how a tool should behave, but to *describe* the characteristics of an existing design tool. After all, the exact behavior of a design tool is already defined in its code. This is essential because it implies that we are actually duplicating information. A tool description is always an abstraction of the true behavior of a design tool. We want to approach the reality as close as possible for characteristics that are important for design flow management and leave characteristics undescribed if they are irrelevant.

We make a distinction between two categories of tool characteristics important for design flow management:

- *data access characteristics.*
  We use the term "data access" in a broad sense. All operations of tools on data, such as reading data, producing data, modifying data and removing data, are considered to be data access. Knowledge about the data access characteristics of design tools is inevitable for design flow management functions such as design tracking and constraint enforcement.

- *tool control characteristics.*
  Tool control information describes how to control the execution of a design tool. This type of information is essential to perform design flow management functions such as design assistance and design automation. For instance, to assist designers during tool invocation, to invoke design tools automatically or to make a running design tool perform some design function, the design flow system should know how to invoke a tool and how to influence the execution of a tool when it is running.

Throughout this chapter, we examine both categories of tool characteristics in more detail. First, we describe how to model the data access characteristics of design tools. After that, we inspect the information that should be included in a tool description for controlling a design tool.

3.2 Modeling Data Access

One of the main problems we face when modeling data access is that it is not always possible to predict the operation of a tool in advance. Although some tools, such as batch tools, always perform the same data accesses, others, such as interactive tools, decide at run-time which data to access and how. This problem makes us search for a model that offers enough flexibility to describe different kinds of tools. However, at the same time we should be careful not to develop a model equipped with flexible but inexpressive constructs. For the development of such a model, we take an incremental approach. After a brief examination of data and data accesses, we define a simple model with little flexibility and then we add more flexibility to this model bit by bit to finally arrive at a model that offers flexible but also expressive constructs for data access characterization.

3.2.1 Data and Data Accesses

To keep the tool model generally applicable, we make minimal assumptions on the data and the way tools access this data. We assume that data is organized into *data entities*, which have a certain *datatype*. This assumption is very general. In our view, it holds for all CAD frameworks mentioned in section 2.2. Figure 3.1 shows the relationship between data entities and datatypes in an OTO-D (see appendix A) information model. It shows that a *DataEntity* has a *DataType*[2].

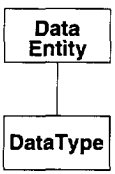


Figure 3.1. Each data entity has a datatype.

With respect to the access of tools to design data, we assume that *tools* perform *tool runs* in which they access *data entities*. Figure 3.2 shows the corresponding OTO-D information model. A *Tool* may be involved in multiple *ToolRuns*. Each

2. One may object that this information model implies that, due to the principle of convertibility, there can never exist two data entities with the same datatype (see also [Bekke92]). This is solved by giving *DataEntity* an extra attribute, for instance *DataEntity-Name*. To keep the information models in this thesis clear and understandable, we do not explicitly show such identifying attributes, but we suppose them to exist where needed.

ToolRun belongs to exactly one *Tool*. A *DataAccess* relates a *DataEntity* to a *ToolRun*. This means that within a *ToolRun* a *Tool* may perform multiple *DataAccesses* on different *DataEntities* and that a *DataEntity* may be accessed multiple times within different *ToolRuns*.

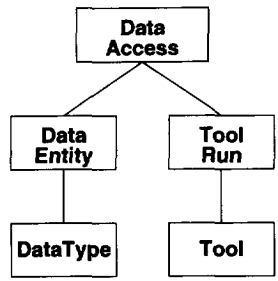


Figure 3.2. A tool may access multiple data entities during a single tool run.

The organization of tools, tool runs and data accesses as specified in the above information model is so general that it applies to most CAD frameworks. For those CAD frameworks that do not conform to this information model, it can be seen as a basic extension necessary to perform design flow management. We use this information model as a starting point for developing our tool model.

3.2.2 Ports

Before developing constructs for data access characterization, we should first decide which characteristics to describe. First of all, we limit ourselves to data access characteristics that can be used for the purpose of design flow management. The information in a tool description should contribute to the goal of design flow management: assisting designers during the design process. Second, we focus on those data access characteristics that have a high chance to be known for a design tool. It makes no sense to develop constructs for data access characteristics that cannot easily be described. For instance, it is not useful to develop constructs to describe *which* data a tool is going to access since this may vary from tool run to tool run. We aim at characteristics that are stable for different tool runs (the *invariants* of a tool). Two data access characteristics that do satisfy these criteria are the *type* of data to access and the *mode* used to access data.

The *type* of data accessed by a design tool is a valuable source of information for a design flow system since it can be used to determine the applicability of design tools and to schedule design tools to generate data of a specific type. For most design tools the type of data on which they operate is known at tool configuration time (in section 3.2.5 we address tools for which this is not the case).

The *mode* used to access data is crucial information for a design flow system since data accesses with different modes may affect the design state in a different way. For instance, to determine the executability of a design tool or to predict the state of design after successful tool execution, it is important to know whether a tool performs read accesses or write accesses or both.

To describe the types of data a tool accesses and the access modes it uses, we introduce the *port* construct. A port describes the access to data of a certain *datatype* and with a certain *access mode*. We do not predefine the possible values for the datatype of a port since this depends on the types of data supported in a particular design process and since we can describe design flow management functionality without knowing the exact values of the datatypes. We suppose that the actual datatypes available in a specific design process can be configured at design flow configuration time.

With respect to the access mode of a port this strategy cannot be followed. Although each CAD framework may support different modes to access the design data, we cannot leave the possible values for the access mode undefined. Since many of the design flow management functions must be defined in terms of the access modes of a tool, we must specify a number of general access modes in our tool model. To keep the concepts generally applicable, we suppose the absolute minimum of access modes: an *input* and an *output* mode. The input access mode describes that design data is accessed to inspect its contents, without modifying it. The output access mode describes the creation of design data.

In figure 3.3 we show the OTO-D information model for the port construct. In this figure and in all subsequent information models the dashed line indicates the boundary between the *run-time part* of the information model, which describes run-time information and the *configuration part* of the information model, which describes configuration information (see also section 2.4.1) The configuration part shows that a *Tool* may have a number of *Ports*, where each port describes the access to design data of a certain *DataType* and with a certain *AccessMode*. Legal values for *AccessMode* are *input* and *output*. The run-time part shows that within a *ToolRun* of a *Tool* multiple *DataAccesses* on *DataEntities* may be performed via (maybe different) *Ports* of this *Tool*. We require that a *DataAccess* performed in the context of a *ToolRun* of a *Tool* refers to a *Port* of that *Tool* and that the *DataType* of a *DataEntity* accessed in the scope of a *DataAccess* equals the *DataType* of the *Port* of that *DataAccess*. These leads to a static constraint, which is expressed in OTO-D as follows:

ASSERT DataAccess ITS ValidDataAccess (TRUE) =
ToolRun ITS Tool = Port ITS Tool AND
DataEntity ITS DataType = Port ITS DataType

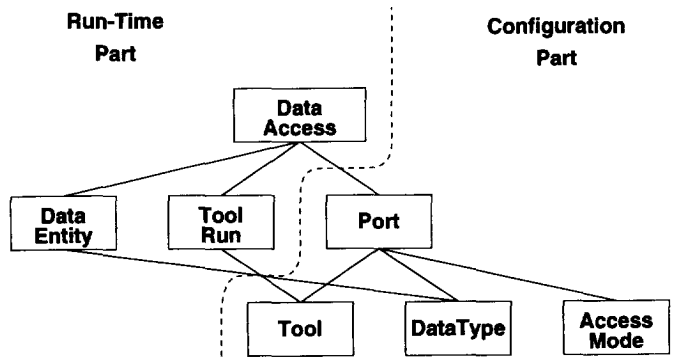


Figure 3.3. A port describes the access to data of a certain datatype and with a certain access mode.

3.2.3 Activities

The tool model presented in the previous section has a number of weaknesses. First of all, a design tool can only be described if it always performs the same number of data accesses with the same characteristics. This makes this model unsuited to describe design tools that can operate in different modes in which they perform different data accesses or to describe design tools that decide at run-time which type of data to access, with which mode and how many times. Unfortunately, such design tools are quite common in complex design processes. The exact operation of a design tool may depend on command line options, user interaction, the state of its input data or notifications from other tools. Especially the operation of graphical, interactive design tools via which designers may perform many different design operations during a single tool run, cannot be predicted in advance. To enable the description of tools for which the data accesses cannot be determined at tool configuration time, we should make the tool model more *flexible*.

The second weakness of the tool model presented in the previous section is that it supports the description of data access characteristics of tools at the coarse-grain level of tool runs only. However, to effectively assist designers with their complex design tasks, the design flow system should have at its disposal more detailed information about the behavior of the design tools. For instance, using an interactive schematics editor, a designer may perform an arbitrary number of

updates on possibly different schematics during one and the same tool run. The successful completion of the schematics editor tells nothing about the number of schematics edited during its execution. The unit of tool operation meaningful to the designer is the update of a single schematic. Another example is a hierarchical layout to circuit extractor that produces a number of circuits that equals the number of layouts in the hierarchical layout description. From the viewpoint of the designer, the production of a single circuit is a meaningful unit of tool operation. We term such a unit of tool operation meaningful to the end-user a *design function* of a tool. Ideally, designers should be assisted at the level of design functions of design tools instead of at the level of tool runs. This is in line with the objective to perform fine-grain design flow management as stated in chapter 1. Therefore, the tool model should offer constructs to describe the capabilities of design tools in terms of its available design functions.

We use the notion of a design function of a tool to solve the problem that the exact operation of a design tool cannot always be known at tool configuration time. We view a tool run as consisting of smaller units of tool operation, which do have well-defined data access characteristics and which correspond to the design functions of a tool. We call such a design function of a tool an *activity* of the tool and we allow a tool to perform any number of its activities any number of times in any order during a single tool run. A tool may have multiple activities each having a number of ports, which describe their data access characteristics.

The refined information model in figure 3.4 shows that a *Tool* may have multiple *Activities* and that each *Activity* belongs to exactly one *Tool*. An *Activity* has a number of *Ports*. The run-time part of the schema shows that during each *ToolRun* of a *Tool*, multiple *ActivityRuns* may be performed and that within an *ActivityRun* multiple *DataAccesses* may be performed on (maybe different) *Ports*. An *ActivityRun* of a *ToolRun* of a *Tool* may only be performed by an *Activity* of this *Tool*. This static constraint is expressed by:

ASSERT ActivityRun ITS ValidActivityRun (TRUE) =
  ToolRun ITS Tool = Activity ITS Tool

The static constraint recognized in section 3.2.2 should be rewritten as:

ASSERT DataAccess ITS ValidDataAccess (TRUE) =
  ActivityRun ITS Activity = Port ITS Activity AND
  DataEntity ITS DataType = Port ITS DataType

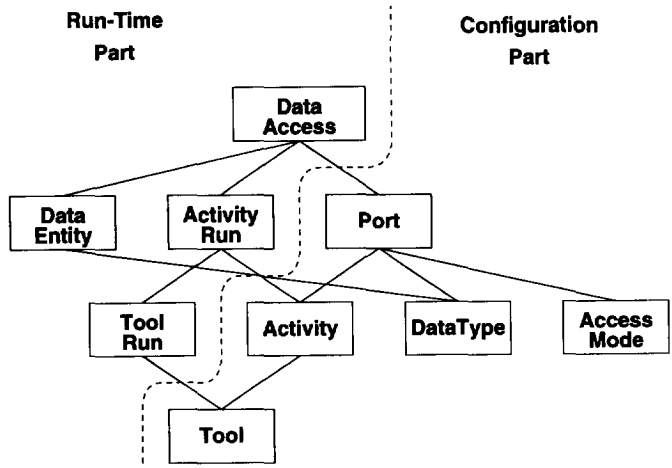


Figure 3.4. Data access characteristics are defined per activity.

Depending on what is considered to be a design function of a tool, the number of activities of a tool and the complexity of an activity may vary considerably. In general, simple design tools that perform straightforward design operations have no more than one activity and complex interactive design tools that may perform a variety of design functions have multiple activities. An activity may describe a complicated access pattern or may be very simple. For instance, it may be desirable to define an activity that only performs one read access if this is considered as a meaningful unit of tool operation of the design tool.

Because a design tool may perform any of its activities multiple times during a tool run, it makes no sense to define two activities for a design tool that do not differ. If a design tool has two activities that are essentially the same, the same design function of this tool is defined twice. We therefore observe the following:

Observation 3.1:

The activities of a design tool have different data access characteristics.

We will use this observation in chapter 6, where we describe the implementation of the concepts for design flow management in the Nelsis CAD framework.

Although the activity concept closely allies to the general observation that many design tools perform a small number of well-defined design functions an arbitrary number of times in an arbitrary order, there are situations where it is difficult to define activities that describe the design functions of a tool as experienced by the designers and at the same time have well-defined data access characteristics. In the next two sections we describe two of such situations and their solutions.

3.2.4 Multiplicity of Data Access

If the activity construct is the only way to describe tools in a flexible way, this may lead to the definition of *phoney activities*: activities that do not correspond to a unit of tool operation meaningful to a designer. For instance, if an optional data access of a tool is modeled by two activities, one with and one without a port describing this data access, these activities are considered to be different design functions of the tool and treated as such in the design flow system. However, the difference between these activities may be unimportant from a designer's point of view. Also, if each variation in the data accesses of a design tool leads to the definition of a new activity, the number of activities of a tool may become very large. In the extreme, it may be necessary to split up a design tool into activities that perform one data access, in which case they may no longer correspond to the design functions of the design tool. Since we want activities to describe the design functions of a tool as experienced by the designers, we need, in addition to the activity mechanism, other constructs for flexible data access characterization.

An effective method to further increase the flexibility of the tool model is to allow ports to describe *optional* data access. Optional data access of an activity may be performed or not during an activity run. Since each data access is administered at run-time, it is always known whether data has actually been accessed or not. The only implication of configuring optional data access is that the information at the disposal of the design flow system is less precise. The design flow system is uncertain about whether a certain data access will be performed or not during an activity run. Therefore it will be less precise in its prediction of the executability of design tools and the state of design after successful execution.

Except for optional data access, it may also happen that a tool performs one type of data access *multiple* times, while this number cannot be specified at tool configuration time. To enable the generation of a tool description for such tools, it is useful to describe these multiple data accesses via one port. This mechanism is useful for data accesses that are actually unimportant for design flow purposes or impossible to describe in detail. For instance, it can be used to describe design tools that perform a large number of read accesses to traverse the design hierarchy of their input design data. The individual read accesses are of little concern to the end-users. In addition, since the number of read accesses cannot possibly be known at tool configuration time, it is convenient to describe them all together via a single port.

The specification of *optional* or *multiple* data access says something about how many times a data access is performed. To describe this we extend a port with a *multiplicity* attribute, which may have the following values:

- *required*:
  Data access of the specified data access mode on data of the specified datatype is always performed exactly once during an activity run. This mode can be used to describe important data accesses in an unambiguous way.

- *optional*:
  Data access of the specified data access mode on data of the specified datatype is performed once, or not at all. The ability to describe optional data access in a tool model increases the possibility to define activities that really correspond to design functions. Note however that it is still possible to define an optional data access via two activities: one with a port describing the data access and one without. This may be useful to distinguish between design functions of a tool that differ in one data access only.

- *multiple*:
  Data access of the specified data access mode on data of the specified datatype may be performed any number of times. This value can be used for data accesses that are more or less unimportant for design flow purposes or impossible to describe in detail.

In figure 3.5 we show the extension to the information model. A *Port* has a *Multiplicity*, which specifies how many times a data access with a certain *AccessMode* is performed on design data entities of a certain *DataType*. Possible values for *Multiplicity* are *optional*, *required* and *multiple*.

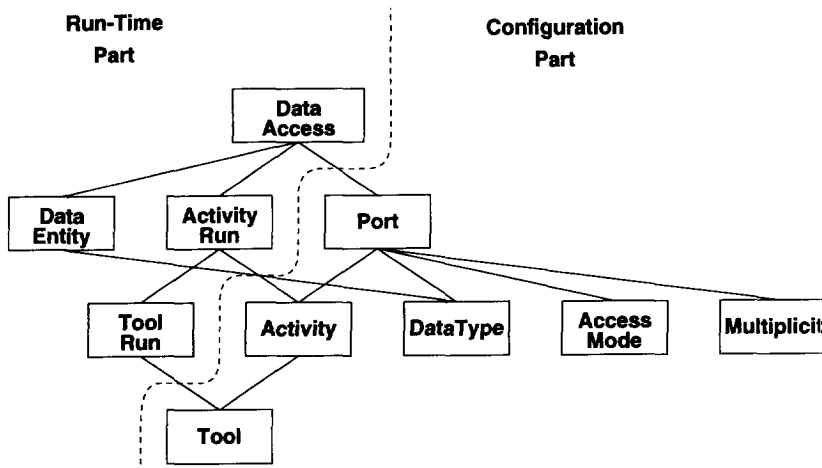


Figure 3.5. The multiplicity of a port describes how many times a data access is performed.

3.2.5 Generic DataTypes

For most design tools the type of design data they access is encoded in their tool code (or in their wrappers). However, it may be that this information is not encoded in the tool itself but that it is derived from the environment in which the tool operates. For instance, this is the case if the type of data is specified via the command line. In such situations, the type of data to access is determined at run-time and cannot be known at tool configuration time. This behavior is typical for *generic tools*, which may operate on many different types of data. A simple example of such a generic tool is an ascii text editor. It cannot be determined at tool configuration time whether such an editor will be used on EDIF descriptions or VHDL data. This is determined by the type of data on which it is invoked.

Using the tool model as presented so far, the only way to describe a generic tool is to generate an activity for each valid combination of datatypes the tool can access. In the example of the ascii text editor, this would result in an activity for editing EDIF descriptions and one for editing VHDL data. However, generic design tools are typically used in a number of different design environments, which each may have their own set of datatypes defined. Since we like to allow tools to be described independent of the design environments in which they are being used, the definition of a number of different activities to completely specify the data access characteristics of generic design tools is not an ideal solution. It would be better if we could explicitly describe the generic behavior of such design tools.

Although at tool configuration time we cannot possibly know all the datatypes of the data entities that a generic tool will access, we may know a more generic datatype indication. For instance, the generic text editor mentioned above can only be used on ascii text. This brings us to the idea that we should offer constructs to define the type of data to be accessed in a generic way so that the binding to an actual datatype of a data entity is postponed until the tool is used in a specific design environment. We explain this binding mechanism in more detail in section 4.2.1. To describe the type of data to be accessed in a generic way, we use a (class) hierarchy of datatypes, where a parent datatype may bind to one of its children datatypes, and the datatype of a data entity is always a leaf datatype. Using this mechanism, ports of generic tools can refer to non-leaf datatypes and ports of non-generic tools can refer to leaf datatypes. For instance, the description of the datatype a tool accesses can vary from very general, such as "text", to more specific, such as "expanded layout file" or "c file that is checked by lint". For the generic text editor mentioned before we can specify that it only handles ascii text correctly and cannot be used on binary files. The EDIF and VHDL datatypes should then be children of the ascii text datatype.

Figure 3.6 shows an information model that supports the hierarchical specification of datatypes. A *DataTypeHierarchy* relationship relates a *Parent-DataType* to a *Child-DataType*. Due to the use of a datatype hierarchy the second static constraint mentioned in section 3.2.3 needs to be relaxed. Instead of requiring that the *DataType* of a *DataEntity* of a *DataAccess* equals the *DataType* of the *Port* of this *DataAccess*, the first *DataType* may also be a descendent of the second *Datatype*. This complicated static constraint cannot be expressed easily in OTO-D. Another static constraint that results from the use of a datatype hierarchy is that the *DataType* of a *DataEntity* must be a leaf in the datatype hierarchy. This implies that for each *DataType* of a *DataEntity* there is no *DataTypeHierarchy* relationship in which this *DataType* is a *Parent-DataType*. This constraint is expressed in OTO-D in the following way. First, we have to extend the *DataType* object type with an attribute *IsParent*, which administers whether a *DataType* is a parent in the *DataTypeHierarchy*:

EXTEND DataType WITH IsParent =
    ANY DataTypeHierarchy PER Parent-DataType

Then, we assert that the *DataType* of a *DataEntity* is not a parent:

ASSERT DataEntity ITS ValidDataEntity (TRUE) =
    DataType ITS IsParent = FALSE

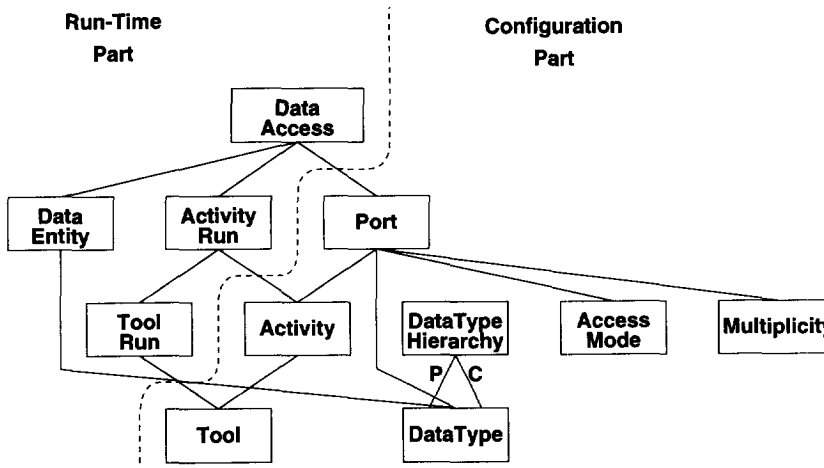


Figure 3.6. The use of a datatype hierarchy supports the definition of generic tools.

3.3 Modeling Tool Control Information

To assist designers during tool invocation, to automatically start up design tools or to make a running design tool perform some specific design function, a design flow system must know how to *control* the execution of the design tools. Since the methods to control a tool may vary from tool to tool, this kind of information should be described in the tool descriptions and the tool model should offer constructs for this. In this section we develop such constructs. We concentrate on two types of tool control characteristics in particular: *tool invocation characteristics* and *inter-tool communication characteristics*. Tool invocation characteristics have to do with the start up of a design tool and describe for instance how to run a design tool in a specific mode or how to run it on some specific type of data. Inter-tool communication characteristics describe how to influence a running design tool to make it perform some specific design function.

3.3.1 Tool Invocation

If the invocation of a design tool would be no more than simply specifying its name, there would be no need to extend the tool model with constructs to describe the tool invocation characteristics of design tools. However, the reality is that many design tools have a large number of "options" or "parameters" that can be specified to run the tool in a certain way. Especially in design environments that contain a large number of design tools each having their own specific options and parameters, it may be difficult for designers to always make the right choice. In such design environments it is useful to feed the specific knowledge about the options and parameters of the tools into the design flow system so that it can assist designers during tool invocation. In addition, this knowledge enables the design flow system to track the invocation of the design tools in a detailed way, to advise designers during tool invocation based on decisions they took earlier and to predefine certain options and parameters for a specific design process.

In our view, the earlier mentioned Tool Encapsulation Specification (TES) [CFItes92] offers much of the functionality needed to describe tool invocation characteristics of design tools. Since we do not want to reinvent the ideas underlying TES, we adopt some of its more useful concepts into our tool model. One of these concepts is the ability to describe a tool invocation command line of a tool in terms of *command line arguments* (arguments for short) instead of via a single string. The fact that this feature has also been used as a criterion for comparing a number of design flow systems in [Kleinfeldt94], strengthens us in our opinion that this is indeed an important characteristic of a design flow system.

To make the tool model suited for the description of tool invocation information in terms of arguments, in the next two sections we extend our tool model with a (TES-alike) argument construct. First we address the definition and administration of arguments and then we describe how arguments can be predefined for a certain design process.

3.3.1.1 Arguments

For the description of the syntax and structure of a tool invocation command line, it is useful to split up the command line into a number of arguments. Each argument specifies the syntax and semantics of part of the command line. TES offers extensive constructs for describing tool invocation arguments. For instance, arguments can be typed to specify what kind of value[3] they represent and they can be accompanied by an interval that delimits the possible values. An argument may only be assigned a value of the right type that lies within the specified interval. To describe the complicated relationships between different parts of a command line, arguments may refer to other arguments. To generate a valid tool invocation command line, each argument should have a value assigned of the right type and satisfying the conditions that were specified for this argument. With each assignment the command line becomes more precisely known. The actual command line to be used upon tool invocation is determined by the complete set of argument assignments.

To describe tool invocation characteristics of design tools in terms of arguments, we extend the tool model with an *argument* construct. We do not describe the specification of the characteristics of an argument in detail since TES offers extensive capabilities for this. We concentrate on the interplay between TES-alike arguments and the constructs offered in our tool model. Using the idea that a tool invocation command line consists of arguments, a tool in the tool model may have any number of arguments.

One kind of argument is important to describe in more detail: arguments that refer to data. These arguments are special in the sense that the value assigned to them is determined by the data on which the design tool operates. For instance, one of the arguments of a design tool operating on a certain data entity may evaluate to the name of this data entity. Since our tool model supports the description of data

3. Actually, TES-arguments can be multi-valued. For simplicity, in this tool model we only use single-valued arguments.

access characteristics of tools, we can make this relationship between arguments that refer to some data access and the description of this data access more explicit. Since data access is described by the ports of the activities of a tool, arguments that refer to data may be related to the ports of the activities of the tool it belongs to. We call this relationship a *port assignment.*

The command line that is actually used to initiate a particular tool run is characterized by a number of arguments with a value assigned to them. This information should be administered to give designers the ability to browse through previous command lines and to build default command lines based on earlier tool invocations. To let a design flow system inspect and use the command lines stored, it is necessary to administer the command line used per argument.[4] This increases the usability of previous command lines for tool scheduling and for advising designers.

The information model of figure 3.7 shows that each *Tool* may have multiple *Arguments* for its command line specification. An *Argument* has a number of attributes (not shown) that define its characteristics, like its type and syntax. A *PortAssignment* relates an *Argument* of a *Tool* to one of the *Ports* of one of the *Activities* of this *Tool* to indicate that this *Argument* depends on the data accessed via this *Port.* This leads to the following static constraint:

ASSERT PortAssignment ITS ValidPortAssignment (TRUE) =
  Port ITS Activity ITS Tool = Argument ITS Tool

Each *Argument* can be used (*UsedArgument*) in a *ToolRun* a number of times with different *Values* assigned to it.

4. An actual implementation may choose to store command lines as strings and decompose them into arguments when needed. However, this is an implementation issue. The important thing is that used command lines must be decomposed into the arguments and their assigned values, to be of use to the design flow system.

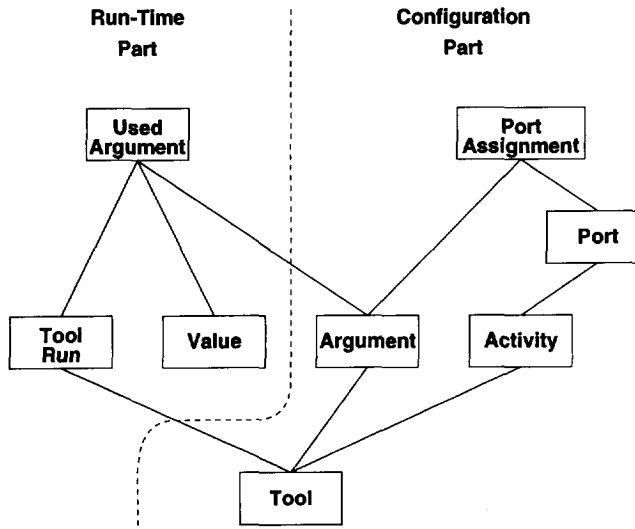


Figure 3.7. Arguments specify the command line of a tool.

3.3.1.2 Predefining Arguments

Since we aim at a design flow system that manages the design process at the level of activities (fine-grain design flow management), it is better to describe tool invocation information at the level of activities than at the level of design tools. In other words: we like to be able to invoke an activity rather than a tool. However, this is only possible for design tools that allow the activity they perform to be specified from the command line. For such tools, it is useful to define which values should be assigned to arguments to let a tool perform a particular design activity. This implies that we must allow certain arguments to be *predefined* per activity.

In figure 3.8 we added this extension to the information model. Since, with this extension, values can be assigned to arguments for two purposes: for the administration of used arguments and for predefining arguments, we create an object type *InstantiatedArgument* (*Inst. Argument*). An *InstantiatedArgument* is an argument with a value assigned to it. An argument can be instantiated multiple times for different values. A *PredefinedArgument* assigns an *Instantiated-Argument* to an *Activity*. The *Def.Type* (*Definition Type*) defines additional information on the predefinition of the value (for instance, whether it may be overruled or not). For the administration of the command line actually used we offer a comparable construct. An *UsedArgument* is redefined to assign an *InstantiatedArgument* to a *ToolRun*.

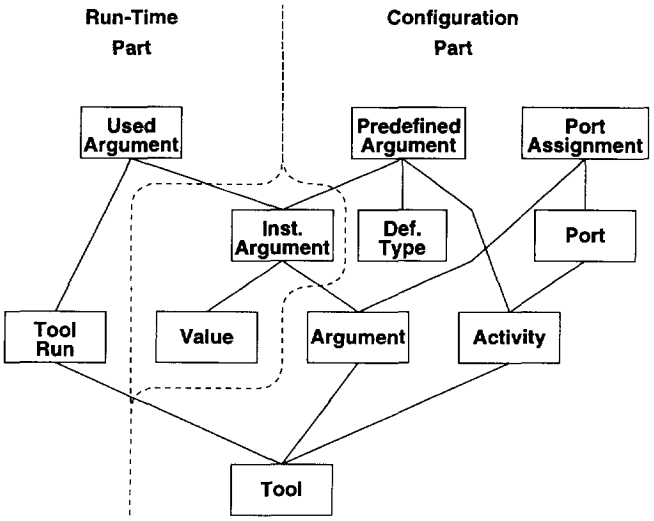


Figure 3.8. The value of arguments can be predefined for activities.

There is one sneaky problem in this information model. For the administration of the arguments used, there must exist an instantiated argument with the right value. If such an instantiated argument does not exist, it should be generated. Since it is unlikely that instantiated arguments exist for all possible values of an argument, some instantiated arguments need to be generated at run-time. This makes that instantiated arguments are not purely configuration information. However, it is not purely run-time information either, since for configuring the predefined arguments of an activity, instantiated arguments must be defined at configuration time. Actually, some instances of this object type are created at configuration time, others are created at run-time. Therefore, we define the instantiated arguments and values to be in an intermediate area (see figure 3.8). Instances of object types in this area can be generated at run-time as well as at configuration time.

3.3.2 Inter-Tool Communication

If design tools have the ability to communicate via some inter-tool communication protocol, it may be useful for a design flow system to know the messages these tools understand and their meaning. The design flow system may use this knowledge to influence the execution of a running design tool. In particular, it would be very useful if the design flow system knows what messages can be sent to a running tool to let it perform a specific activity. For instance, a running simulator could be instructed to simulate a just created schematic description.

A necessary condition for this way of controlling tools is that these tools understand messages defined in some inter-tool communication protocol. One of the most popular standards for inter-tool communication is the ToolTalk standard from SunSoft [ToolTalk91a]. The ToolTalk standard supports the definition of *message sets* defining a protocol for communication among a number of inter-operating applications. CFI has developed its own standard for inter-tool communication, the Inter-Tool Communication Programming Interface [CFIitc92]. The CFI standard is more specific in the sense that it focuses on inter-tool communication in electronic design environments. The generality of the ToolTalk standard as opposed to the CFI standard is emphasized by the fact that the CFI standard can be implemented using ToolTalk (see [ToolTalk91b]). We do not examine the different standards in more detail. The issue here is that if design tools are sensitive to some kind of inter-tool communication message and if this message can be used to initiate an activity of this tool, then we would like to make this information available to the design flow system. We call such an activity an *inter-tool communication activity.*

With respect to the information model this means that we should add an an *Inter-Tool Communication Message* (*ITC Message*) attribute to an *ITC activity*, which is a special type of *Activity*. This is shown in figure 3.9.

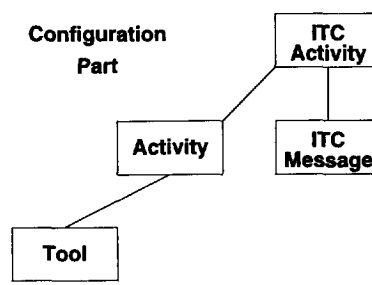


Figure 3.9. An ITC activity can be initiated by an ITC message.

3.4 Comparison with Other Approaches

In this section we compare the tool model presented with a number of other approaches that address the tool modeling phase.

3.4.1 TES

In the CFI Tool Encapsulation Specification (TES) [CFItes92], data access is described mainly by its *direction*, which may be INPUT, OUTPUT or INOUT and by the *condition* under which data access is performed. The condition is a boolean expression that may refer to any other description of data access or command line argument in the TES specification. In TES, there is no way to specify the type of data that is accessed. The TES INPUT and OUTPUT data directions map quite smoothly onto our input and output access modes. The activity concept is not supported in TES: there is one TES description per tool that describes all data access characteristics of the design tool. However, this is compensated to some degree by the more extensive facilities to describe under what circumstances a particular data access is performed. In our view, the absence of data typing and the activity concept makes TES unsuited to describe the data access characteristics of design tools for the purpose of design flow management. We think the tool model presented in this chapter offers more advanced facilities for this purpose.

With respect to tool control information TES is probably the most extensive language that currently exists. The power of TES is its expressiveness and its simplicity. It is relatively simple to make a TES description for a complicated command line structure and to build a software module that reads TES specifications and helps designers to invoke their design tools based on these specifications. For instance, two systems that use TES in this way are the Nelsis CAD framework [Bosch93] and AutoCap [Câmara94]. Unfortunately, TES also has a number of weaknesses. Some of the constructs are ambiguous, there is a typing problem and there is a problem concerning multi-valued strings. Due to the many indistinctnesses in the specification (version 1.0.0), TES has not evolved into a successful, generally accepted standard. For a detailed examination of the problems encountered in TES and the proposals put forward to attack these problems we refer to [CFItesdisc93].

The problems in TES version 1.0.0 brought CFI to investigate whether the standard of the Common Desktop Environment (CDE) [CDE93] could replace TES. At the time of writing, this investigation is still ongoing. In our opinion, the switch to CDE does not solve the real problems in describing tool control characteristics of design tools. Our feeling is that the CDE-constructs for describing tool control information are more restricted than those offered by TES.

However, due to this simplicity CDE contains less indistinctnesses, which increases its chances for a global acceptance.

Another striking difference between CDE and TES is that CDE offers an extensive language, which can be used to program the tool invocation of a design tool. This is different from the approach taken by TES. In TES tool characteristics can be *defined*, and the resulting definition can be inspected by any tool invoker that understands TES. CDE on the other hand offers a language to program a wrapper for a design tool and this wrapper can be *executed* to invoke the design tool. In our opinion the CDE approach does not enrich a design flow system with any knowledge about the design tools itself. The wrapper program simply adds another layer between a CAD framework and the design tools. We think that in order to enrich a design flow system with knowledge of the design tools, the TES approach should be taken. We will not go into the differences between CDE and TES any further here. For a more detailed overview of our contribution to the TES versus CDE discussion, see [CFItesdisc93].

3.4.2 Roadmap

In the Roadmap approach [Hamer90] a three level information model is presented. Level 1 of this information model addresses the description of tool characteristics. As described in [Hamer91] the Roadmap object types *FlowType*, *Pin* and *PinType* map quite cleanly to our object types *Activity*, *Port* and *DataType*. However, there is a difference between the relation between a *Tool* and a *FlowType* in the Roadmap model and a tool and an activity in the tool model presented in this chapter. The *FlowTypes* of a *Tool* in the Roadmap approach correspond to the ways the tool can be run (e.g. by specifying certain command line switches or invocation options). This is different from an activity, which is a unit of tool operation that may be performed any number of times in any order during a single tool run.

The Roadmap model distinguishes between input and output *pins*. This can be compared with our input and output access modes. A difference between the Roadmap model and the tool model presented here is that the Roadmap model has no specific constructs to describe the data accesses of design tools in a flexible way such as the ability to describe multiple data access via one port or to describe the type of data a tool accesses in a generic way.

3.4.3 Hercules Task Schema

The Hercules Task Management system [Brockman91, Sutton93], uses a *task schema* to define dependencies between design entities, which may be both tools and data descriptions. This model does not describe the data access characteristics of design tools in detail. It mainly models the dependencies between tools and data. The interesting thing about the task schema is that tools that are created during the design process can be described as well. For instance, this can be used to describe the characteristics of a simulator compiler that produces an executable simulator.

Although the description of tool characteristics of tools that are created during the design process may be useful in some design applications, we have a slightly different view on this issue. In the task schema, the characteristics of a tool that will be generated during the design process, must be known before the tool is produced. This may be the case for a simulator produced by a simulator compiler. However, this does not necessarily need to be the case for other programs generated during the design process. For instance, in general, we cannot predict the characteristics of a tool that is produced with a c-compiler. In our opinion, if the characteristics of a tool that is produced during the design process are known, the generated tool can be configured anyway. However, if the characteristics of such a tool cannot be predicted in advance, the producer of the tool should generate a tool description for the produced tool. This implies that the configuration information is adjusted during the design process. This can be done with the task schema approach, as well as with the tool model presented in this chapter.

3.5 Conclusion

To let a design flow system perform the design flow management functions mentioned in chapter 1, it should have at its disposal knowledge about the design tools available to the designers. Ideally tool programmers should, aside from an executable program, also produce a *tool description* describing the essential data access and tool control characteristics of the tool in question. In this chapter we presented a model for describing such characteristics of design tools for the purpose of design flow management.

In the first part of this chapter we presented a model to describe the data access characteristics of design tools. Central in our approach is the notion of an activity as a design function of a design tool, which may be performed any number of times during a run of that tool and which has well-defined data access

characteristics. Design tools that may perform a number of totally different design functions, or that may perform a design function an unpredictable number of times, can be described conveniently using the activity concept. The ability to describe the operation of design tools in terms of design functions of tools rather than in terms of tool runs closely allies to the fact that many of today's design tools are long running design applications which may be used to perform many different design operations per tool run.

An important characteristic of the model presented is its *flexibility*. Constructs such as activities, multiplicity and a datatype hierarchy make it possible to generate a detailed description of data access characteristics that are known at tool configuration time or a more vague description of data access characteristics that cannot be known in detail at tool configuration time. This makes the tool model presented suited to describe the essential characteristics of design tools operating in real design processes. Instead of describing ideal tools in ideal design environments, the tool model presented can be used to describe real tools in real design environments.

An interesting observation with respect to the flexibility of the tool model is that, in the end, it is up to the design flow configurator to use the flexible constructs or not. However, the price to be paid for using these constructs is that the knowledge about the design process at the disposal of the design flow system is less precise, which will decrease the assistance power of the design flow system. So it is up to the design flow configurator to choose an optimum between vague tool descriptions and optimal designer assistance.

For describing tool control information, we pursued the policy to describe tool characteristics as much as possible per activity. We created the opportunity to define how to initiate a particular design activity of a design tool by specifying the right values for the arguments on its command line or by sending it an inter-tool communication message.

In the next chapter we examine how tool descriptions can be used to build a design flow for a particular design process.

4

Design Flow Modeling

4.1 Introduction

To assist designers during a complex design process, a design flow system must, apart from knowledge about the capabilities of the individual design tools, also have an idea of how the design tools can be combined to bring the designer closer to a possible design solution. For instance, it should have detailed knowledge about the relationships between the different design tools, about the preferred or non-preferred order of execution of the design functions of the design tools and about the optimal use of the design tools with respect to the current design objectives. In other words, it should have knowledge about the *structure* of the design process. As described in chapter 1, this type of information is encoded in a *design flow*. In this chapter we develop a number of constructs to describe the structure of a design process in a design flow. Together with the constructs presented in the previous chapter, these constructs define the *design flow model* to be used in our design flow system.

To describe the structure of the design process, we like a design flow to express design *constraints*, such as constraints on the execution of design steps or constraints on the relationships among data involved in these design steps, as well as information on the *organization* of the design process, such as the organization of basic design steps in more abstract design phases or the relationships between design data accessed in the course of the design. This brings us to the following three characteristics of a design process to be described in a design flow. First of all, we develop constructs to describe the dependencies between the design functions offered by the different design tools. Second, to express the structure of a design process in terms of more abstract design tasks, we develop constructs to organize design flows in a hierarchical way. Third, we offer constructs to describe (constraints on) the relationships among data involved in a design process. In addition, we define a number of rules that define the dynamics of a design process described in a design flow.

4.2 Modeling Dependencies

An essential question that must be answered before we can get into modeling dependencies between design tools is what kind of dependencies to model. Dependencies one can think of are:

- *data dependencies*:
  A data dependency between two design tools indicates that one tool may operate on data produced by another tool.
- *temporal dependencies*:
  A temporal dependency between two design tools specifies a constraint on the execution order of these tools.
- *control dependencies*:
  A control dependency between two design tools specifies that one tool controls the operation of another tool.

To judge the necessity to support each kind of dependency in the design flow model, we have to keep the goal of a design flow system in mind: to assist designers during the design process.

We feel that knowledge on the data dependencies among design tools does contribute to the assisting power of a design flow system. Many design tools have very specific data access requirements and these are not always completely determined by the type of data the tools access. For instance, it may be that an EDIF parser can read all types of EDIF descriptions except for descriptions generated by one particular tool. In this case the type of data (being an EDIF description) does not completely specify whether the EDIF parser can correctly read it. If this type of dependencies between design tools can be described in a design flow, the design flow system can inform designers in more detail about the possible design steps. Therefore, we conclude that it is useful to offer constructs to explicitly describe data dependencies in a design flow.

There is a strong coherence between data dependencies and temporal dependencies among tools. If tool A needs data from tool B then tool B must be executed before tool A. So a data dependency implies a temporal dependency. However the opposite does not hold: if tool A must be executed before tool B then tool B does not necessarily need data from tool A. So a temporal dependency does not imply a data dependency. This means that temporal dependencies may specify constraints on the design process that cannot be described using data dependencies. For instance, it may be desirable to demand that in order to run an expensive simulation, first a check of the input description must be performed.

Since this is a constraint on the execution order of tools that does not follow directly from the data requirements of the individual tools, it cannot be modeled using data dependencies. Therefore, we feel that our design flow model should offer constructs to describe temporal dependencies as well.

With respect to control dependencies, we feel it is not very useful to describe them in a design flow. The fact that one tool controls the execution of another tool does not say anything about the design operations performed by these tools, nor does it say anything about the applicability of these tools to satisfy some design goal. Therefore, we have the feeling that the specification of control dependencies between tools does not improve the capabilities of the design flow system and we do not examine them any further.

Suppose we have a design environment with a tool description for each tool describing the data access characteristics of the tools. Using this information it is possible to derive what we call an *unrestricted design flow*, which describes all data transfer that is "technically" possible. The unrestricted design flow combines maximum freedom of design with a minimum of design constraints. Each more restricted design flow can be derived from the unrestricted design flow by:

- removing data dependencies.
  A technically possible data transport is prohibited.
- adding temporal dependencies.
  An extra constraint is added to the design flow.

This brings us to the observation that by supporting both data dependencies as well as temporal dependencies it is possible to model the design process in detail. Data dependencies as well as temporal dependencies between tools carry a considerable amount of valuable information for the designer and therefore we demand that it must be possible to model data dependencies as well as temporal dependencies in a design flow. In the next sections we develop a number of constructs to describe these two types of dependencies in a design flow.

4.2.1 Data Dependencies

To guide designers correctly through the design process, a design flow system should make sure that the execution of the design tools does not conflict with the data access characteristics of the tools and with the data production and consumption constraints that hold in a specific design process. Therefore, a design flow model should offer the ability to describe data dependencies between design tools. In the previous section, we already noticed that starting from the tool descriptions of a set of tools, data dependencies can be derived based on the type

of data these tools access. In this way all "technically possible data transfer" can be described. However, not all data transfer that is possible from a technical point of view should be allowed in a design flow. First of all, it is possible that a tool has no problem accessing data produced by one tool, but cannot correctly handle data produced by another tool, although both tools produce data of the same datatype. Second, when tools can read each others data without any problem, this is not always preferred from a design flow point of view. In a specific design process one may want to prohibit a technically possible data transfer. This implies that the tool model should offer constructs to explicitly describe the producer-consumer relationships among tools.

Since data access characteristics of design tools are described by the ports of their activities, data dependencies can be described by connecting the ports via which data transfer is allowed. Because in a design environment multiple design tools may operate on data of the same datatype and multiple design tools may produce data of the same datatype, we choose to describe allowed data transfer by connecting a number of ports that describe data production to a number of ports that describe data consumption. For this, we introduce the *channel* construct. A channel connects a number of *producer* ports to a number of *consumer* ports and specifies that data produced by the producer ports may be accessed via the consumer ports. We notice that the allowed data transfer could also be modeled using a number of directed edges each connecting one producer port with one consumer port. However, in that case there is no notion of a bundle of data transfer with common characteristics. The use of channels stresses the fact that multiple producer ports may transfer the same kind of data to multiple consumer ports.

To prevent channels from describing data transfer that is technically impossible, the datatypes of ports connected to the same channel should be compatible. If there is no datatype hierarchy in the tool model, this simply means that all ports connected to the same channel should have the same datatype. If the tool model does support a datatype hierarchy, ports also have compatible datatypes if they have a common descendent datatype. In this way, a port of a generic tool, which refers to a non-leaf datatype in the datatype hierarchy, may *bind* to a more specific datatype. Although this mechanism may be used to bind ports of generic tools to leaf datatypes, it does not guarantee that all ports in a design flow are actually bound to a leaf datatype. For instance, if two ports of the same non-leaf datatype are connected to the same channel, no binding takes place. For such situations, there is no other way to know the actual datatype to be accessed than to wait until the tools actually execute.

Although the implications of the use of a datatype hierarchy are still limited in this section, the use of a datatype hierarchy would seriously complicate a clear explanation of the concepts for design flow management to be presented in the subsequent chapters. Therefore, from now on we ignore a possible datatype hierarchy in the tool model. We stress however, that all concepts to be presented can be adapted to work properly with hierarchical datatypes as well.

With respect to the access modes of ports connected to a channel, there are no constraints. The access mode of a port connected to a channel defines whether it is a producer (output) or a consumer (input) port of that channel. With respect to the multiplicity of ports there is no constraint either. Since a channel defines *allowed* data transfer of a certain datatype, it is perfectly valid to connect ports with multiplicity *required, optional* and *multiple* to the same channel.

In order to describe data dependencies referring to ports of activities, we extend the information model for tool characterization with a channel construct. In figure 4.1 we show the extended information model. *Ports* can be *Link*ed to *Channels*. A *Port* may be *Link*ed to multiple *Channel*s and a *Channel* may be *Link*ed to multiple *Port*s. Since, due to the absence of a datatype hierarchy, a channel may only connect ports of the same datatype, a *Channel* has an attribute *DataType* and there is a static constraint that the *DataType* of *Ports* linked to a *Channel* equals the *DataType* of the *Channel*. This is expressed in OTO-D as:

ASSERT Link ITS ValidLink (TRUE) =
  Port ITS DataType = Channel ITS DataType

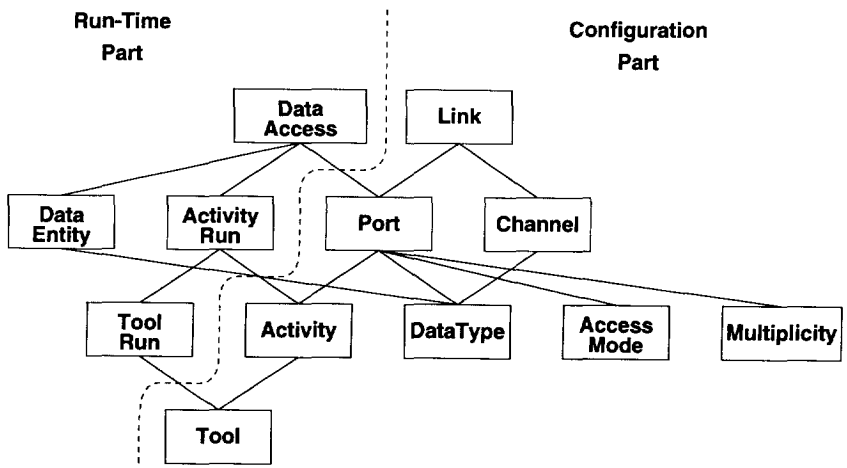


Figure 4.1. A channel describes allowed data transfer.

Now that we have created constructs to describe allowed data transfer in a design flow, we can more specifically define the conditions for data access. An activity is allowed to access a data entity via an input port, only if that port is connected to the port via which that data entity has been produced. We define this more formally. We use the following notation: the name of an object type in upper case (for instance LINK) denotes the set of instances of that object type (e.g. all links in a design flow configuration). We use italic lower case names to denote an instance of an object type (for instance $l \in$ LINK). We use the name of an object type as an operator on an instance to denote the value of the object type of that particular instance (for instance, we write Port(l) for the port of link l, which corresponds to the OTO-D query "GET Link 'l' ITS Port"). First we define under which conditions two ports are *connected.*

Definition 4.1:

Two ports $p_1, p_2 \in$ PORT are *connected*, ($conn(p_1, p_2)$), iff
AccessMode(p_1) = *output*, AccessMode(p_2) = *input*, and
$\exists l_1, l_2 \in$ LINK with Port(l_1) = p_1, Port(l_2) = p_2 and
Channel(l_1) = Channel(l_2).

We use the notion of connectivity of ports to define allowed data access.

Definition 4.2:

An activity $a \in$ ACTIVITY is allowed to access a data entity
$de \in$ DATAENTITY iff $\exists p_1, p_2 \in$ PORT, $da \in$ DATAACCESS with
$conn(p_1, p_2)$, Activity(p_2)=a, Port (da) = p_1 and DataEntity (da) = de.

Figure 4.2 shows a graphical representation of an example design flow. Activities are represented as rectangles, ports as small squares on the left (input) or right (output) side of the activity rectangles, channels are drawn as solid lines and links are drawn as dashed lines. The design flow consists of a number of activities, a,b,c,d and e, and a number of data dependencies between these activities. Activities a and b each have one output port and the activities c,d and e each have one input port. Channel $ch1$ connects the output ports of activities a and b to the input ports of activities c and d. Channel $ch2$ connects the output port of activity b to the input port of activity e. The data dependencies described by channels $ch1$ and $ch2$ specify that activities c and d may read data only if it is produced by activity a or b and that activity e is allowed to read data produced by activity b only.

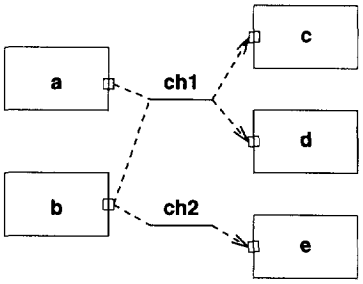


Figure 4.2. Channels describe allowed data transfer between ports.

As explained in [Hamer91] the model presented here differs from the well-known electrical netlist model in the way it connects ports to channels. Where in the netlist model each port can be connected to a single channel only, our model allows a port to be connected to multiple channels. An advantage of this model is that it allows different consumer ports to be connected to different sets of producer ports for which the intersection is non-empty. For instance, in figure 4.2 both activities *d* and *e* may access data produced by activity *b* but only activity *d* may access data produced by activity *a*.

Starting from a set of tool descriptions, the design flow system can automatically generate the *unrestricted design flow* mentioned before, by defining channels between all ports of all activities with compatible datatypes. This unrestricted design flow may be a starting point for the definition of a more restricted design flow. As an example, consider figure 4.3. The channels and links in this design flow describe all data transfer that is allowed with respect to the datatypes of the ports. In this figure we show the datatype of each port ($dt1$, $dt2$, $dt3$ or $dt4$) and we use solid lines for channels as well as for links. From now on, we show the difference between links and channels in figures only if this is important to be recognized.

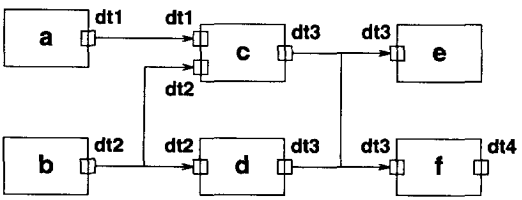


Figure 4.3. An unrestricted design flow.

4.2.2 Temporal Dependencies

As described in section 4.2, in certain design environments there are constraints on the execution order of the design tools that do not originate from the data requirements of tools. These constraints cannot be expressed using the constructs presented so far. As an example, figure 4.4 shows a design flow consisting of a layout editor, a design rule checker and a layout to circuit extractor. In this example it may be desirable to require that each layout has been checked before it may be extracted. However, since there is no real data transfer between the design rule checker and the extractor, this constraint is not a data dependency, it is a *temporal dependency.*

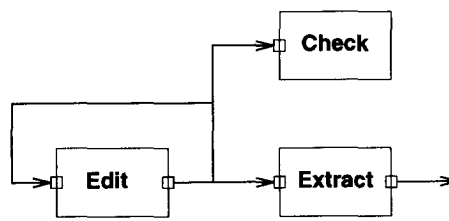


Figure 4.4. Example design flow.

If, in the above example, the design rule checker would always generate a log file, which would be consulted by the extractor, the above constraint would naturally follow from the data transfer between the two tools. This brings us to the idea that the temporal dependency in the above example can be seen as a special type of data dependency where both the producer activity (check) as well as the consumer activity (extract) do not access the data. We use this idea to model temporal dependencies with a minimal extension to the design flow model. This has the advantage that to the end user, we can use one formalism for presenting data dependencies as well as temporal dependencies.

We see a temporal dependency as a data dependency without data access. In contrast to a data dependency, where both the producer activity and the consumer activity actually access the data, in the case of a temporal dependency there is no data access at one or both sides. Since data that is not produced cannot be accessed, a dependency without data access on the producer side and with data access on the consumer side cannot be satisfied in any way. Therefore, temporal dependencies are either dependencies with no data access on the consumer *and* producer side, or absent data access on the consumer side only. Figure 4.5 gives an overview of the type of dependencies with respect to the existence or absence of data access on the producer and the consumer sides.

Consumer / Producer	Data Access	No Data Access
Data Access	Data Dependency	Temporal Dependency
No Data Access	Impossible	Temporal Dependency

Figure 4.5. The type of dependency is determined by the existence or absence of data access on the producer and consumer side.

Since temporal dependencies can be viewed as data dependencies without physical data transfer, we extend the existing design flow constructs to express temporal dependencies. In order to describe temporal dependencies using channels and ports, we must be able to express that a port does not describe data access, but acts as a placeholder for a temporal dependency. This special property is modeled by defining a port to be *virtual*. Since it makes no sense to define dependencies that connect virtual producer ports to non-virtual consumer ports (see figure 4.5), we have a constraint that virtual producer ports may be connected to a channel only if all consumer ports connected to that channel are virtual as well. We extend the information model for virtual ports. In figure 4.6 the *Virtuality* of a port defines whether a port describes data access or not. Legal values for the *Virtuality* are *virtual* and *non-virtual*.

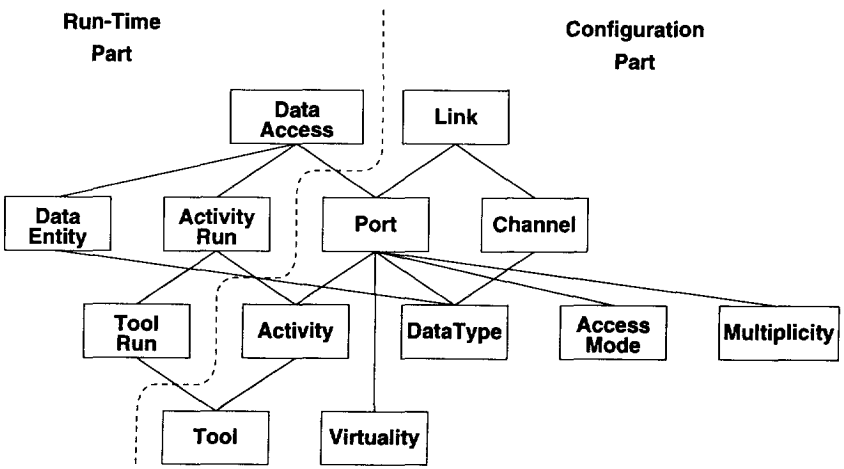


Figure 4.6. The virtuality of a port describes whether data is actually accessed via that port or not.

Now that we have defined how to describe temporal dependencies, we investigate how to evaluate them at run-time. Since there is no data access performed via virtual ports, this is different from evaluating data dependencies. The evaluation of a data dependency for a certain data entity is unambiguous in the sense that the data entity for which the data dependency must hold is specified upon data access. When evaluating a temporal dependency it is not always known with respect to which data or activity run the temporal dependency must hold. To explain the difficulties in evaluating temporal dependencies, consider the example design flows in figure 4.7. To distinguish virtual ports from non-virtual ports, virtual ports are drawn dashed.

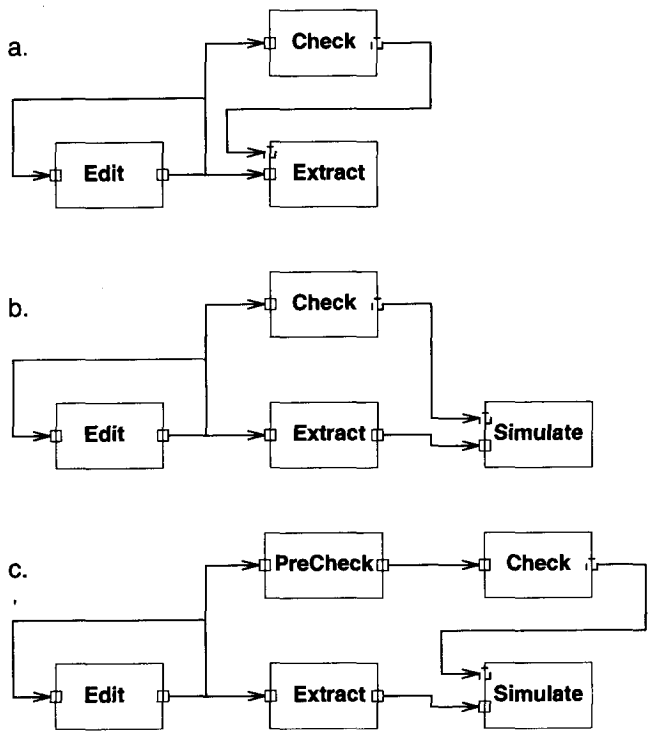


Figure 4.7. Example design flows with temporal dependencies.

Figure 4.7.a shows the example design flow of figure 4.4 extended with a temporal dependency. The temporal dependency between the checker and the extractor specifies that a layout check must be performed before the extractor may be executed on the layout. In this example, the scope of the temporal dependency is clear. It relates to the layout that must be checked and on which the extractor works.

Figure 4.7.b contains a more complicated example of the use of a temporal dependency. The temporal dependency between the checker and the simulator specifies that a check must have been performed before the circuit extracted from the layout can be simulated. However, it is essential that this check has been performed on the layout from which the circuit to be simulated has been derived. Therefore, when evaluating the temporal dependency, it has to be determined from the design flow to which data the temporal dependency relates. In this example this is unambiguous: for each circuit there is always at most one layout from which it has been derived. However, it is possible to define a design flow where the scope of a temporal dependency is not so clear. An example is shown in figure 4.7.c.

In figure 4.7.c, a layout check is performed in two steps. First a pre-check is made and then a detailed check is performed. A layout may be pre-checked multiple times with different options. In this example, the scope of the temporal dependency between the detailed check and the simulator is unclear. To which of the pre-check results does it relate? Must all pre-check results have been checked in a detailed way, or just one of them?

The above examples show that for evaluating a temporal dependency the scope of this dependency must be derived from the design flow. In approaches that require a design flow to be attached to a data entity (usually called a *cell*), such as the JCF approach [Liebisch92], this problem does not exist. In such approaches it is always clear with respect to which data a temporal dependency holds. However, a disadvantage of these approaches is that it is impossible to execute an activity in a flow on data from different cells. Since we think that such a restriction does reduce the applicability of the concepts, we do not follow these approaches and we let the design flow system inspect the design flow to find out with respect to which data a temporal dependency holds.

We like the design flow system to determine the scope of a temporal dependency by searching for *related data* starting from the virtual input port in question. The search for related data may yield multiple data entities. It must be decided whether a temporal dependency must hold with respect to *all* related data entities or only with respect to *one* of them. It is our experience that the requirement that at least one data entity can be found for which the temporal dependency holds, offers enough modeling power to define complicated design environments. Therefore, when evaluating a temporal dependency, we require that there is at least one related data entity for which the temporal dependency is satisfied.

We define the evaluation of a temporal dependency more formally. First we define the *relatedness* of data. We recognize that, in general, data may be related in many ways, for instance through hierarchical relationships, version relationships or equivalence relationships. The relatedness we address here is the relatedness of data entities that arises from their involvement in activity runs. For instance, we define two data entities to be related if they have been accessed during the same activity run. For evaluating temporal dependencies it may be necessary to search for related data via multiple activities. Therefore we define data entities to be related via multiple activity runs as well:

Definition 4.3:

Two data entities $de_1, de_2 \in$ DATAENTITY are *related*, $(rel(de_1, de_2))$, iff $\exists da_1,..,da_{n+1} \in$ DATAACCESS, $n \geq 1$ with DataEntity$(da_1) = de_1$, DataEntity$(da_{n+1}) = de_2$ and ActivityRun(da_i) = ActivityRun(da_{i+1}), $1 \leq i \leq n$. If $n=1$ we say that de_1 and de_2 are *directly related*.

We use the notion of data relatedness to define how to evaluate a temporal dependency. For evaluating a temporal dependency a search to related data has to find out to which data entities the temporal dependency relates. A temporal dependency is satisfied if the search to related data finds at least one data entity for which the temporal dependency holds starting from another input port of the activity with the virtual input port of the temporal dependency. Formally:

Definition 4.4:

A constraint expressed by a port $p \in$ PORT, AccessMode(p) = *input*, Virtuality(p) = *virtual*, is satisfied iff $\exists da_1, da_2 \in$ DATAACCESS with Activity(Port(da_2)) = Activity(p), AccessMode(Port(da_2)) = *input*, DataEntity$(da_1) = de_1$, DataEntity$(da_2) = de_2$, $rel(de_1, de_2)$, and

(i) Port$(da_1) = q$ and $conn(q,p)$

or

(ii) Activity(Port(da_1)) = ac and $\exists q \in$ PORT with Virtuality(q) = *virtual*, Activity$(q) = ac$, $conn(q,p)$

Part (i) in the above definition is necessary to handle situations where a producer port of the temporal dependency is non-virtual and part (ii) to handle situations where such a port is virtual, which is the case in the example design flows of figure 4.7. It is clear that the evaluation of a temporal dependency may take quite a lot of effort in complicated design flows. However, it is our experience that the depth of the related data search is restricted in most design flows. It is the responsibility of the design flow configurator to use this mechanism in a sensible way. If used with care, the ability to define temporal dependencies is a powerful instrument for describing complicated design environments.

4.3 Design Flow Hierarchy

Although the design flow constructs presented so far allow a detailed and unambiguous description of a design process in a design flow, in comprehensive design processes such a design flow may easily grow so complex that it needs some further organization. Especially in design processes with many activities and many dependencies between these activities, it may be useful to organize activities into higher level design tasks and to view the design flow at this more abstract level. An effective way to do this is to use *hierarchical design flows* (sometimes called *flow nesting* [Kleinfeldt94]). A hierarchical decomposition of the design flow can be used to create multiple levels of abstraction by defining abstract design tasks and to reduce the complexity of the design flow at the user interface level by grouping certain parts of the design flow together.

To allow a hierarchical description of the design flow, we introduce a *flow graph* as a generalization of an activity: a *flow graph* is either an activity, or consists of several other flow graphs. We describe a design flow hierarchy by parent-child relationships between flow graphs. These relationships describe the instantiation of *child flow graphs* into their *parent flow graphs*. Activities are flow graphs without children (the leafs in the flow graph hierarchy). If a flow graph is not an activity we call it a *compound flow graph*. A *root flow graph* is a flow graph without a parent. The graph of parent-child relationships is a directed acyclic graph.

With the introduction of a flow graph hierarchy, we have to refine the way ports can be connected to channels. It must be possible to describe data and temporal dependencies between compound flow graphs and their children flow graphs. The connectivity model must be extended to be able to connect uninstantiated ports of compound flow graphs and instantiated ports of children flow graphs to channels.

In figure 4.8 we show the extensions to the information model. A *FlowGraph* is a generalization of an *Activity*. Only *FlowGraphs* that are *Activities* belong to a *Tool*. A *FlowHierarchy* defines the instantiation of a *Child-FlowGraph* into a *Parent-FlowGraph*. The *Link* object type has been split into two different object types. A *PortInstLink* links an instantiated *Port* (a port in the context of a certain *FlowHierarchy*) to a *Channel*. In a hierarchical design flow a *Channel* does always belong to a particular *FlowGraph*. A *PortLink* links an uninstantiated *Port* to a *Channel*. In this information model we have the static constraint that each *Activity* is a leaf in the flow graph hierarchy. This constraint is expressed in OTO-D in the following way (note the similarity with the constraint expressed in section 3.2.5):

EXTEND FlowGraph WITH IsParent =
ANY FlowHierarchy PER Parent-FlowGraph

ASSERT Activity ITS ValidActivity (TRUE) =
FlowGraph ITS IsParent = FALSE

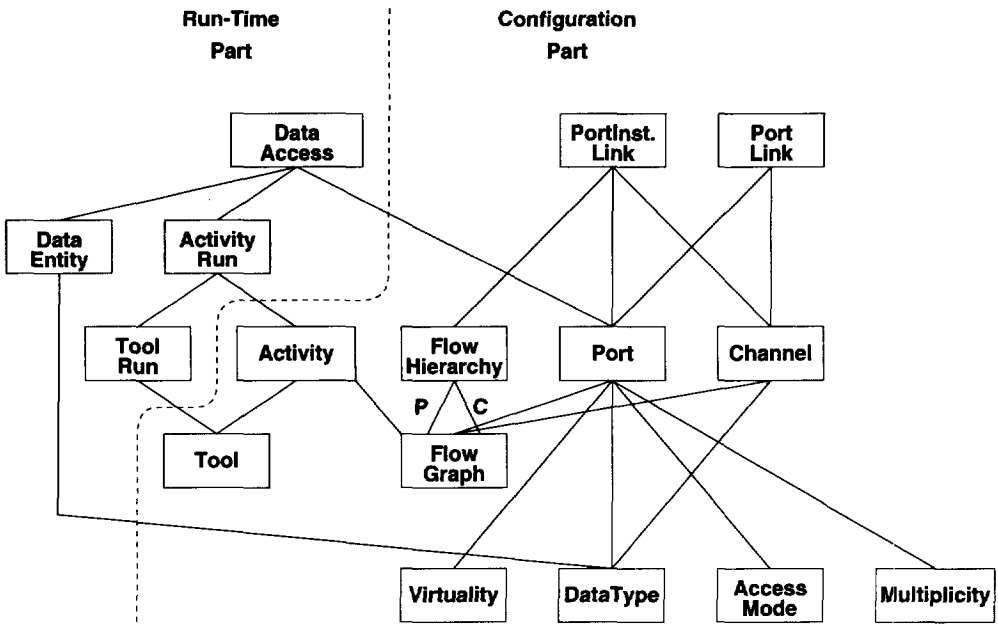


Figure 4.8. Information model for hierarchical design flows.

Figure 4.9 gives an example of a hierarchical design flow. The design flow hierarchy graph in 4.9.a shows that flow graph *d* is instantiated in several other flow graphs, that flow graph *e* is instantiated more than once in the same parent flow graph and that multiple root flow graphs (*root* 1 and *root* 2) can be defined. The definition of multiple root flow graphs in one design flow hierarchy can be used to switch from one design flow to another. The choice for a specific root flow graph implies the choice for a specific design flow. Using this mechanism, different designers may define their own root flow graph with their own favorite design flow. Figure 4.9.b shows the compound flow graph *c* and its children flow graphs *a* and *b*, which are activities. Channel *ch* 1 is connected to the instantiated ports of activities *a* and *b* via port instance links *pil* 1 and *pil* 2 and to the uninstantiated ports of the compound flow graph *c* via port links *pl* 1 and *pl* 2.

Although the use of port instance links and port links is a correct way of modeling the connectivity in a hierarchical design flow, we introduce one more

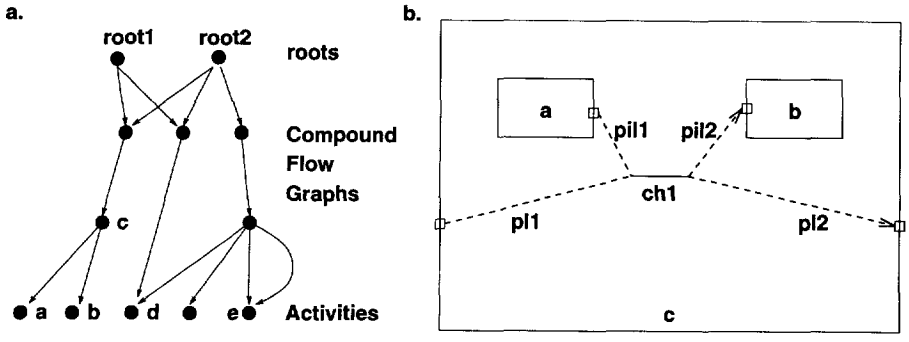


Figure 4.9. Example hierarchy of flow graphs.

simplification. We introduce the constraint that each uninstantiated port is linked to exactly one channel. Since it is still possible to link instantiated ports to multiple channels, this does not restrict the ability to describe dependencies. The advantage of this connectivity model is that it is simpler. Instead of two types of links, we have one type of link left. In figure 4.10, we show this alternative way of modeling connectivity. Note that since the *FlowGraph* of a *Port* can be derived via its *Channel*, we have removed the relationship between *Port* and *FlowGraph*. We will use this connectivity model in the rest of this thesis.

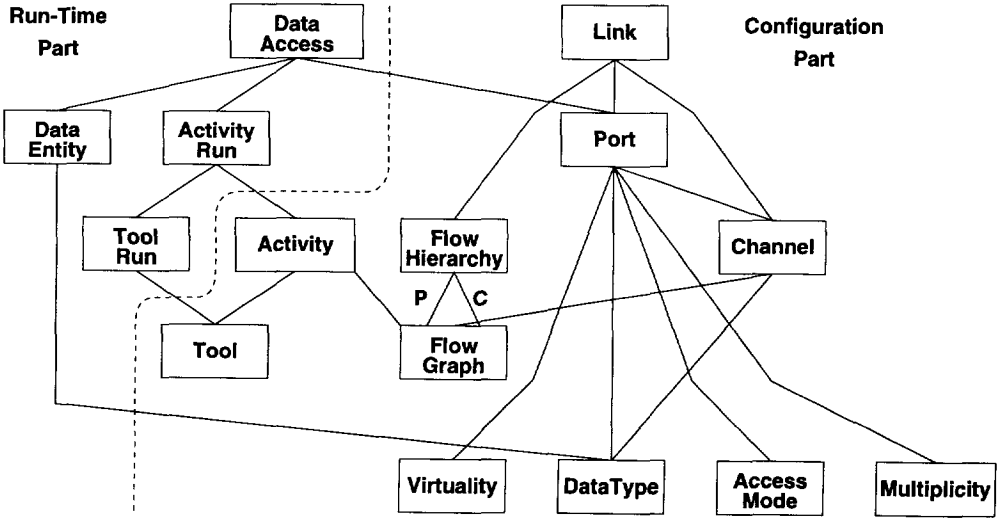


Figure 4.10. An alternative connectivity model.

Figure 4.11 shows the compound flow graph c of figure 4.9.b using the alternative connectivity model. It shows that channel $ch1$ is connected to the instantiated ports of activities a and b via links $l1$ and $l2$ and to the uninstantiated ports of the compound flow graph c directly.

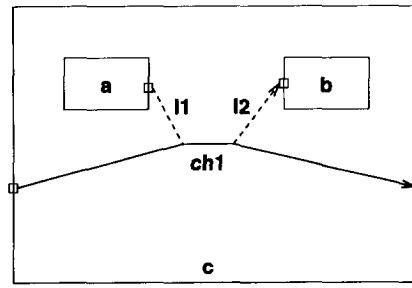


Figure 4.11. An uninstantiated port is linked to exactly one channel.

Due to the addition of hierarchy to the design flow model, we have to refine definition 4.1. Ports of activities may also be connected via channels that describe dependencies at the compound flow graph level. Since the flow nesting may have any depth, two ports may be connected via multiple channels describing dependencies at multiple hierarchical levels. First we define the relationship *adjacent* for ports and channels in a hierarchically organized design flow:

Definition 4.5:

A port $p \in$ PORT and a channel $ch \in$ CHANNEL are *adjacent*, $(adj(p,ch))$, iff

AccessMode(p) = *input* and Channel(p) = ch

or

AccessMode(p) = *output* and $\underset{l \in \text{LINK}}{\exists}$ [Port(l) = p, Channel(l) = ch].

A channel $ch \in$ CHANNEL and a port $p \in$ PORT are *adjacent*, $(adj(ch,p))$, iff

AccessMode(p) = *output* and Channel(p) = ch

or

AccessMode(p) = *input* and $\underset{l \in \text{LINK}}{\exists}$ [Port(l) = p, Channel(l) = ch].

Two ports $p_1, p_2 \in$ PORT are *adjacent*, $(adj(p_1,p_2))$, iff

$\underset{ch \in \text{CHANNEL}}{\exists} [adj(p_1,ch), adj(ch,p_2)]$.

Figure 4.12 shows three situations in which holds $adj(p_1,p_2)$

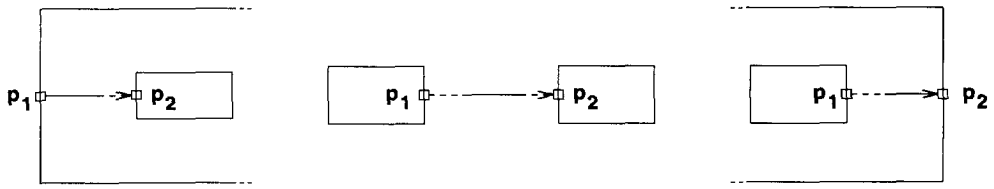


Figure 4.12. Three situations in which holds $adj(p_1,p_2)$.

Using the adjacent relationship we can refine the relationship *conn* (definition 4.1). With $\Gamma_r^*(x)$ we mean the reflexive and transitive closure of x with respect to the relationship r:

Definition 4.6:

Two ports $p_1,p_2 \in$ PORT are *connected*, $(conn(p_1,p_2))$, iff $p_2 \in \Gamma_{adj}^*(p_1)$.

In a hierarchical design flow an activity may be instantiated multiple times. Because we offer a detailed mechanism for describing data dependencies and temporal dependencies between instantiated activities, it may be that some instances of an activity can be executed on some input data and other instances of the same activity can not be executed on the same input data. For instance, in figure 4.13 activity d is instantiated two times in different compound flow graphs. In compound flow graph f it can only access data produced by activity b and in compound flow graph g it can only access data produced by activity c. This implies that it matters which activity instance executes. Only activity instances that are allowed to access data should run.

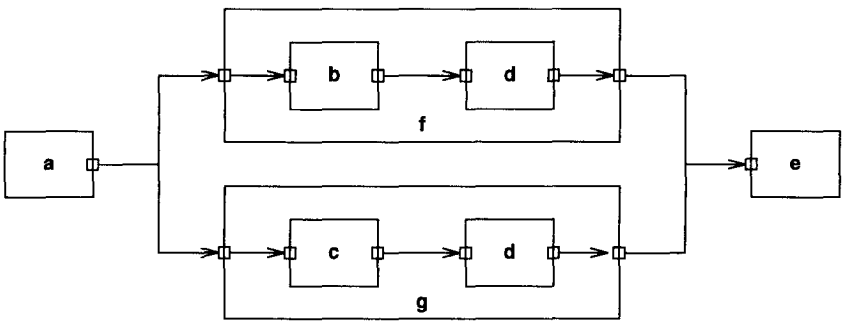


Figure 4.13. Activity d is instantiated multiple times in different compound flow graphs.

This example shows that it is important that the design flow system administers which instantiated activities have been executed and which have not been executed. Therefore, we extend the run-time part of the information model with a construct to describe which instances of an activity have been executed within an activity run. For this, we have to realize that the object type *FlowHierarchy* does only describe a one-level instantiated flow graph. It is unsuited to identify an activity instance completely. This is clarified in figure 4.14. The design flow hierarchy graph in figure 4.14.a shows that activity c is instantiated two times in flow graph b which is, on its turn, instantiated two times in flow graph a. The corresponding tree with flow graph instances is shown in figure 4.14.b. Instance $c\,1$ of activity c cannot be distinguished from instance $c\,3$ of activity c using flow hierarchy relationship $fh\,3$. Activity instance $c\,1$ can only be identified by the full path of flow hierarchies to the root ($fh\,3$,$fh\,1$).

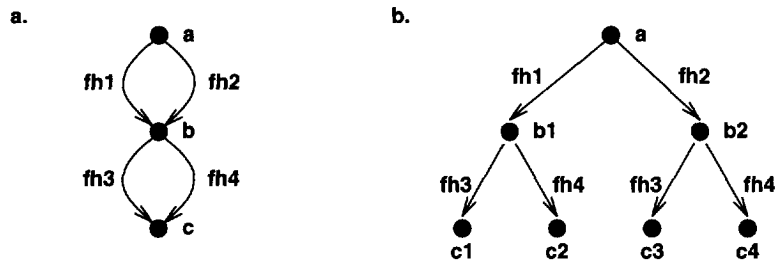


Figure 4.14. A design flow hierarchy that requires the full path of flow hierarchy relationships to identify the activity instances uniquely.

Attentive readers may object that we used the *FlowHierarchy* object type before to link instantiated ports to channels. However, for that purpose a one-level identification suffices. For the connectivity of ports of children flow graphs to the ports of their parents, we only have to identify ports within their parent flow graphs, since it is not allowed to use channels to connect ports of flow graphs that differ more than one level in the flow graph hierarchy. For the identification of activity instances for the administration of activity instance runs this is not enough. We have to uniquely identify the activity instances that have been executed. Therefore we introduce an *ActivityInstance* object type to identify one particular instance of an activity. To describe one element of the full path of flow hierarchies to the root flow graph an *ActivityInstanceDefinition* relates a *FlowHierarchy* to an *ActivityInstance*. An *ActivityInstanceRun* administers the run of one *ActivityInstance* in the context of an *ActivityRun*. These extensions are shown in figure 4.15. The following static constraint must hold:

ASSERT ActivityInstanceRun ITS ValidActivityInstanceRun (TRUE) =
ActivityRun ITS Activity = ActivityInstance ITS Activity

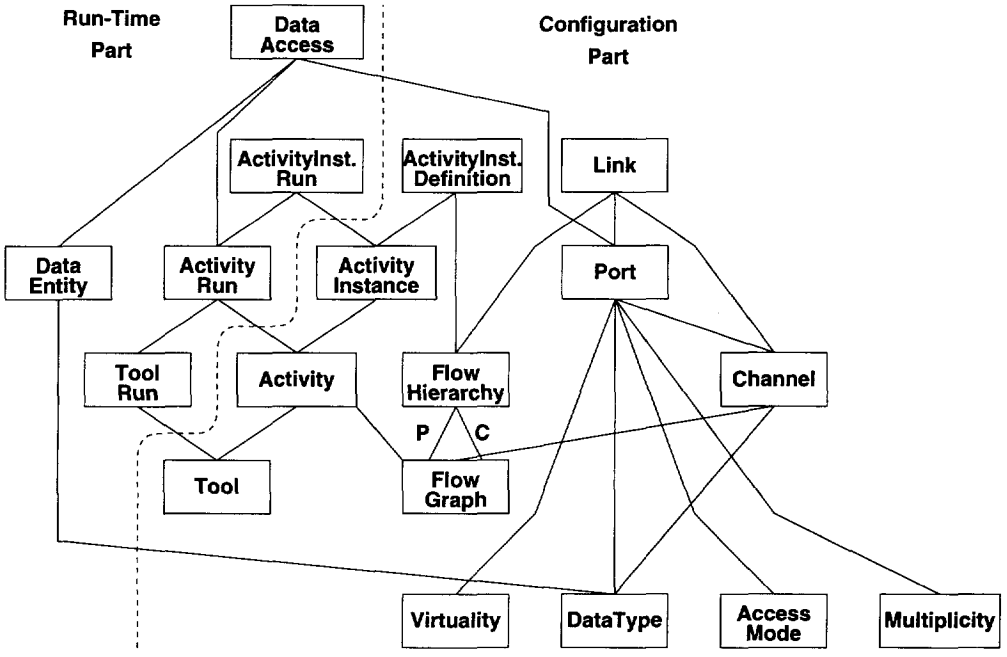


Figure 4.15. An ActivityInstanceRun administers a run of an ActivityInstance.

In chapter 3 we described how arguments can be predefined for activities. It may be convenient to extend this mechanism one step further and to allow the definition of arguments for children flow graphs in the context of a parent flow graph. In this way, one and the same activity can be configured to perform a slightly different task in different parts of the design flow. This is modeled in figure 4.16. A *HierarchyArgument* assigns an *InstantiatedArgument* to a *FlowHierarchy*.

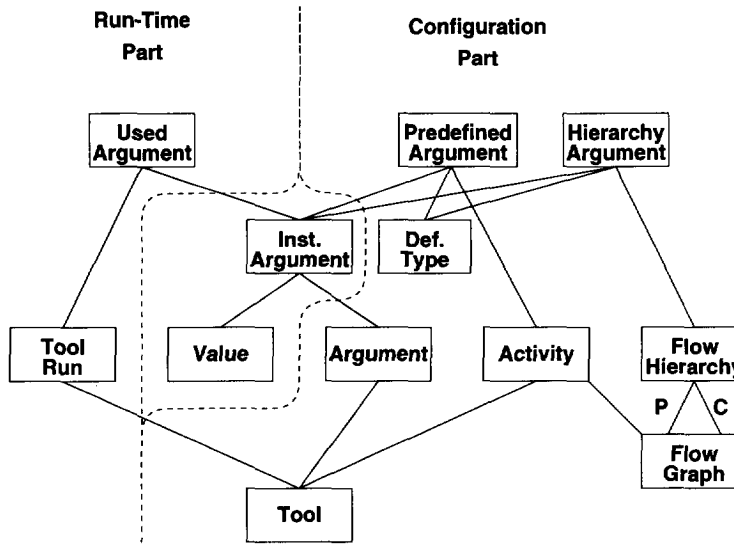


Figure 4.16. The value of an argument can be predefined for children flow graphs in a specific parent flow graph.

4.4 Port Relationships

A design flow configuration defines the organization of the design process for a particular design application. Except for defining activities of tools, data access characteristics of activities, data and temporal dependencies between activities and the hierarchy of the design flow, a design flow configuration can be exploited to predefine other characteristics of the design process as well. In this section we address a number of design process characteristics that have to do with relationships among data entities involved in activity runs. For instance, in certain design environments it is useful to define a maximum on the number of data entities that are allowed to be derived from one input data entity via a certain activity. Another example is that a tool may only work on data that is related via a hierarchy relationship. In this section we develop constructs to express such properties of the design process in a design flow.

Since activities access their design data via ports, properties of the design process that refer to design data entities accessed in an activity run can be defined via the ports of activities. The generic mechanism to be used for this is a *port relationship*. A port relationship defines a property of the design process concerning data involved in an activity run via two ports. The property to be defined and the type of relationship in question can be defined in the *port*

relationship type. This leads to an extension of the information model with a *PortRelationship* between a *Source-Port* and a *Destination-Port* and a *PRType* that specifies the type of the port relationship. This is shown in figure 4.17.

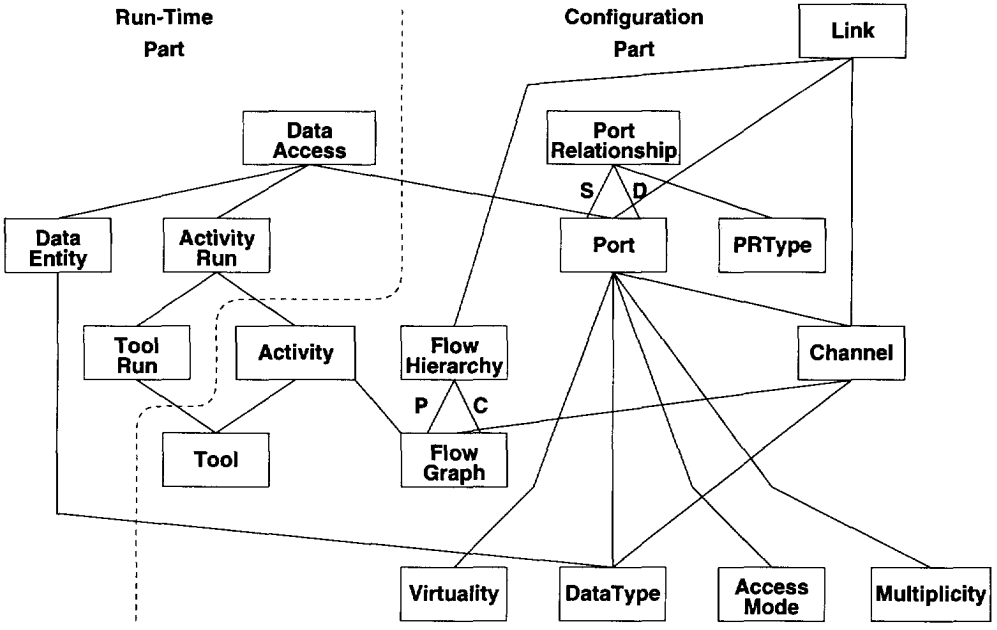


Figure 4.17. A port relationship specifies a property of the design process that refers to the data entities accessed via ports.

A useful application of port relationships is to restrict the number of data entities that may be related via a certain activity. An example of the use of such a constraint is the wish to let an activity not produce more than one output data entity for each input data entity. This is particularly useful for activities of design tools that always give the same result for the same input, regardless of user interaction or command line options. For this type of tools it may be useful to prevent a repeated execution on the same input data, if output data is still valid. This keeps designers from useless executions of activities on the same input data. If output data has been removed there is no related data any more and the execution of that activity is no longer prohibited.

To allow this kind of functionality to be defined in the design flow configuration, we define a port relationship type "*maximum related*" that can be used to restrict the number of related data entities between two ports. Depending on the required behavior of the design flow system, it may be configured that, when the maximum is achieved, the execution of the activity is prohibited or the oldest data entity is

removed automatically. Using a port relationship with such automatic removal, designers are not hampered in their design activities when the maximum number of data entities is reached. However, a disadvantage is that automatic removal of design data is creepy in the sense, that there is always a chance that valuable data is removed.

Another use of the port relationship with respect to the relatedness of data entities, is the propagation of data removal. In certain design applications it may be convenient to remove design data if the data from which it has been derived is removed as well. Since this is only desirable for a subset of activities, it should be possible to configure this behavior explicitly in the design flow configuration. For instance, one may want to remove the results of a simulation if the simulated schematics is removed. On the other hand, it is usually not desirable to remove a circuit if the layout from which it has been extracted is removed. We call the process of automatic removal of design data if design data from which it has been derived is removed "*chained removal*". The administered web of related design data entities can be exploited to propagate removal of design data and the port relationship can be used to configure in which cases a chained removal should be performed. We define a port relationship type *chained removal.* A chained removal port relationship between two ports defines that if data accessed via the source port is removed, that all data related via the destination port is removed as well.

In many advanced design systems, there are more relationships between data entities than just relationships on data relatedness implied by a design flow mechanism. Examples are hierarchical relationships, version relationships and equivalence relationships. In this section we make no further assumptions on the existence of other inter-data entity relationships. In chapter 6 we will see how port relationships can be used to define constraints on the design process with respect to the inter-data relationships managed by the Nelsis CAD framework. The message in this section is that using port relationships a number of detailed properties of the design process can be preconfigured in a design flow configuration. The ability to specify such properties on relationships between data entities in the design flow configuration yields a flexible mechanism for tuning a design system to the specific needs of a design application. It also illustrates the close relationship between design flow management and data management.

4.5 Comparison with Other Approaches

In this section we compare the presented design flow model with a number of other approaches from the literature.

4.5.1 Petri nets

A number of approaches to design flow modeling [Janni86, Bretschnei90, Kupitz92] use Petri nets to describe the data access characteristics of design tools. Tools are described by transitions and data by places. The firing of a transition corresponds to a tool run. All these Petri net-based approaches use some kind of extension of Petri nets (extended Petri nets, predicate transition nets, hierarchical predicate transition nets) because the modeling power of standard Petri nets is too limited.

One of the reasons for the shortcoming of the standard Petri net approach is that Petri nets have a dynamic behavior that does not conform to the operation of design tools. For instance, if a transition fires, the input tokens are removed from their places. For design tools this is typically not appropriate. In general, it is perfectly valid to run a design tool multiple times on the same input. Also, transitions cannot fire if their output places carry tokens. This is unlike a real design situation, where multiple runs can generate many different outputs which are being stored into a big database. Actually, these differences are symptoms of the more fundamental difference that in our approach the state of the design process is determined by the contents of the meta data database of the CAD framework. The complete state of the design process cannot possibly be represented as a single Petri Net marking.

We conclude that, in general, a Petri net approach is not tailored to a description of design processes. If transitions of Petri nets are used for modeling data access characteristics of design tools, the transition rules of Petri nets predefine the dynamic behavior of these design tools in a way that does not conform to the reality of the design tool execution. Also, the transition rules hamper a flexible definition of tool characteristics. The observation that Petri nets are not the right mechanism for modeling tool behavior is confirmed by the fact that the Jessi Common Frame project exchanged their Petri-net based design flows for an activity based approach [Baltes94].

4.5.2 Roadmap

With respect to the Roadmap approach [Hamer90] the design flow model presented in this chapter has comparable constructs for modeling data dependencies and hierarchical flows. For instance, *flowtypes* in Roadmap may be compared with *flowgraphs*, *slots* with *port instances* and a *flow* with a *flowhierarchy*. The main difference between the two approaches with respect to the modeling of data dependencies lies in the connectivity model used. Where we use channels to link multiple output ports to multiple input ports, the Roadmap approach has no notion of a channel and connects inslots (input slots) to outslots (output slots) with the restriction that an inslot is always connected to exactly one outslot. The inability to connect an inslot to multiple outslots is compensated by an extension to the basic Roadmap model, which defines the notion of *pin-compatibility*. Instead of connecting an inslot to multiple outslots to show the alternative producer flows, the designer can choose between one or more pin-compatible variants of the same producer flow. For a detailed comparison between the Roadmap connectivity model and the Nelsis connectivity model, see [Hamer91].

Temporal dependencies between flows are not specifically addressed in the Roadmap approach. Roadmap design flows can be described hierarchically in the same way as presented in this chapter, but an important difference can be found in the administration of runs in a hierarchically organized design flow. Where in our approach we administer the complete path in the flow graph hierarchy of an activity instance that has been executed, the Roadmap approach administers a run on each level of the flow hierarchy.

4.5.3 Hercules Task Schema

A task schema as described in the Hercules approach [Brockman91, Sutton93] mainly describes the data dependencies between tools and data. A difference between the Hercules approach and the design flow model presented in this chapter is the way a design flow is defined. In [Sutton93] a flow is portrayed as a static, fixed sequence of design activities that the designer is required to follow step by step. In order to give the designer greater freedom in choosing how to go about the design, the notion of *dynamically defined flows* is introduced. A dynamically defined flow can be derived from the task schema and contains a fixed sequence of design steps.

The design flow model presented in this chapter does not fit into this view. Since channels may connect multiple producer ports to multiple consumer ports, it is possible to define alternative design strategies in one and the same design flow.

Hence, our design flow approach should be compared with the Hercules task schema rather than with a dynamically defined flow constructed from the task schema. Both approaches allow extensive definition of data dependencies between design activities. With respect to temporal dependencies, no specific constructs are given in the Hercules approach.

4.5.4 The Jessi Common Framework Project

In the approach of the Jessi Common Framework [Liebisch92] a design flow can be defined for each data entity, which is called a *cell*. The dependencies on such a design flow are temporal dependencies. They define the execution order of the design activities on the cell. Since each data belongs to a cell, for which a flow is chosen, it is always clear to which data a temporal dependency in this flow relates. An advantage of this approach is that different design flows can be defined for different cells. However, a disadvantage is that tools configured in a flow cannot be executed on data residing in different cells. We think that this is a major disadvantage and have chosen not to restrict the scope of a design flow to a specific cell. In the design flow model presented design flows define generic characteristics of the design process that are to be applied to all available design data in a project.

4.6 Conclusion

In this chapter we described how, based on a set of tool descriptions for the design tools in a design environment, a number of characteristics of the design process can be configured in a design flow. We started this chapter with the observation that both data dependencies as well as temporal dependencies may contribute to a better description of the design process. Therefore, we have developed a number of constructs to describe such dependencies in a design flow.

Data dependencies describe the allowed data transfer between design activities. We presented a connectivity model that supports the definition of alternative design trajectories in one design flow and that allies to the way designers experience the data relationships between design tools. Rules are given that exactly specify under which conditions an activity is allowed to access a certain data entity with respect to the configured data dependencies.

Although most approaches to design flow management restrict their capabilities to the definition of data dependencies, data dependencies only are not sufficient to describe the design process in detail. To support a wide range of design tools and design applications, it is necessary to offer the ability to define temporal

dependencies in a design flow. Temporal dependencies define additional constraints on the design process that cannot be expressed using data dependencies. Temporal dependencies can be modeled using the same design flow primitives as used for expressing data dependencies, with a small extension for virtual data access. For evaluating a temporal dependency, the design flow system has to find out to which data entities the temporal dependency relates. This *related data* search process is inevitable in a design flow approach that supports the notion of temporal dependencies and that does not a priori bind a design flow to a particular (cell-alike) data entity. A temporal dependency is satisfied if the related data search finds at least one data entity for which the temporal dependency holds.

The third design flow principle is the use of a design flow hierarchy to organize the design flow. Using hierarchically organized design flows, the design flow information can be presented in a structured way, which makes it possible to define high level tasks in compound flow graphs. Also, a hierarchical design flow description can be used in the design flow user interface to present the design flow in orderly parts to the designers.

Finally, we introduced the notion of port relationships that can be used to preconfigure a number of detailed properties of the design process in a design flow configuration. We showed the use of port relationships for the purpose of defining a maximum on the number of data entities that may be related via two ports of a certain activity, and for the purpose of configuring chained removal. In chapter 6, where we apply the presented design flow approach to add design flow functionality to the Nelsis CAD framework, we will show how port relationships can be used to configure constraints on the design process that refer to other inter-data relationships.

In the next chapter, we use the presented tool model and design flow model for the development of concepts for design state presentation and user interaction.

5

The Design Flow User Interface

5.1 Introduction

Although the interaction between a human being and a computer is still very limited compared to the rich and varied communication between multiple human beings, there is one instructive similarity between these two. If a human being has the brightest ideas but cannot transfer his thoughts clearly and unequivocally to other people, he will never be able to convince these people of the value of his ideas. The same holds for a design flow system that is built with the intention to support designers with their complex design tasks. If a design flow system interacts with the designers in an awkward way it can never succeed in guiding them effectively through the design process. The method of user interaction used by a design flow system determines an important part of its success. A design flow system will fail if it has no proper way to communicate with the designers. Therefore, the user interface component of a design flow system, which we call the *design flow user interface*, is a vital element of a design flow system.

In this chapter we develop principles and mechanisms for user interaction to be applied in the design flow user interface. We aim at a design flow user interface that maximally exploits the design flow model described in the previous chapters to assist designers during the design process. To better understand what this means for the design flow user interface, we first take a more detailed look at the basic design actions performed by designers during the design process. We are particularly interested in design actions that have something to do with the interaction between designer and design system. We recognize four different types of such design actions (see also [Wolf94a] and [Bosch93]):

- *browsing through the design state*:
  When browsing through the design state a designer inspects the state of the design process as administered by the data entities, their inter-relationships and the operations performed on these data entities.

- *invoking a design tool*:
  For invoking a design tool a designer must select the tool and the data on which the tool should operate. Furthermore, he must decide how to run the tool. For instance, which options and parameters to use and on which machine to run the tool.

- *operating a design tool*:
  When operating a design tool, a designer interacts with the design tool directly through the user interface of the tool and he inspects the changes on the design state that result from the execution of the tool.

- *organizing design data*:
  Data that has been generated or that has become out of date must be organized, for instance, by changing its status or by manipulating its hierarchical decomposition.

We think that most of the interaction between a design system and a designer can be expressed in terms of these basic design actions. A designer may perform any of these design actions any number of times in any order. To make things even more complex, a designer may perform several design actions concurrently. For instance, in a multi-tasking environment it is a common situation to execute multiple tools simultaneously.

We use this list of design actions for the development of our design flow user interface. We take the overall goal of assisting designers during the design process to mean an efficient execution of these basic design actions. However, the design flow user interface must not only be tuned towards efficient data browsing, tool invocation or data organization, it should also help designers to move from one design action to the next. For instance, it should be as easy as possible to switch from design state viewing to tool invocation and vice versa.

In this chapter we describe a design flow user interface that satisfies the above requirements using only a few simple and intuitive mechanisms. In the next section, we introduce the notion of *flow-based user interaction* (section 5.2). In flow-based user interaction, all user interaction is performed via a design flow based representation of the state of the design process. We describe how the run-time information of a design flow system can be combined with the design flow configuration information to visualize the design state in an intuitive way to the designer. When deriving such a representation, the design flow user interface must be careful not to overwhelm designers with information of which only a small part is of their interest. This is one of the main problems we face when defining a design flow user interface. Mechanisms have to be developed to

analyze the large collection of information stored in the meta data database and to present just the right parts of this collection in an attractive way to the designers. The technique we develop for this, called *flow coloring*, is described in section 5.3. In section 5.4, we address the issue of *design scheduling*. A design schedule can be seen as a description of how to fulfill a specific design goal. We describe how the design flow user interface may support the definition and execution of design schedules.

5.2 Flow-Based User Interaction

CAD framework-based design systems typically offer a set of *browsers* to inspect the state of the design process. Each of these browsers derives a graphical representation of (part of) the meta data. For instance, a version browser displays the version history in a version graph and a hierarchy browser displays the hierarchical decomposition of a design in a tree or graph. Designers typically interact with such browsers to let them show those parts of the design state they are particularly interested in. We call this method of user interaction *meta data browsing*. Figure 5.1 shows an intuitive picture of meta data browsing. Each browser uses a certain *algorithm* to derive an appropriate graphical representation of (part of) the meta data. A designer may give the algorithm *hints* to let it derive a representation that allies to his interest.

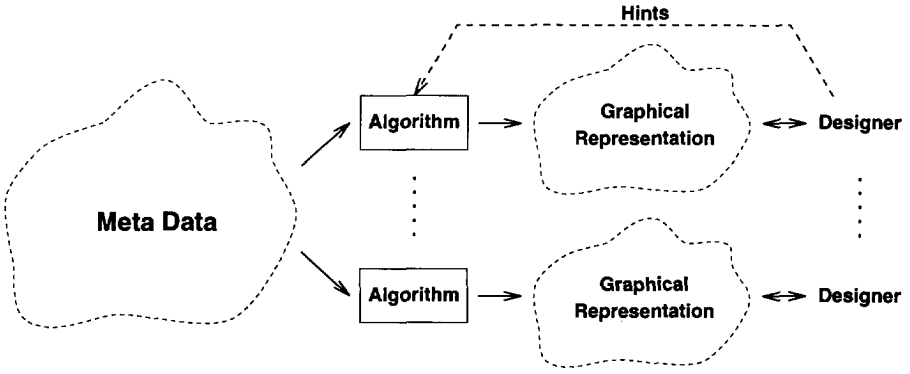


Figure 5.1. Meta data browsing.

Design systems without design flow management usually offer a set of browsers that present a graphical representation of the data entities and their interrelationships (see for instance [Gedye88] and [Bingley90]). Since these representations do not show the relationships between data and tools, a designer cannot see, for instance, which tool produced a certain data entity or which other

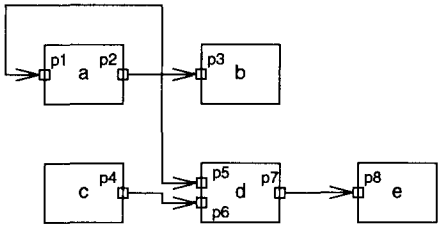
data entities were produced by this tool. This is uncomfortable, since many design decisions are based on these relationships. For instance, the decision to invoke a certain design tool on certain data may depend on the state or history of this data and the validity of a data entity may depend on the result of the execution of a certain tool on this data entity.

In a design flow system that has at its disposal a description of the design process in a design flow and that maintains the design state in the run-time information as described in the previous chapters, it becomes possible to let a user interact with the design system via a graphical representation that shows the intricate relationships between tools and data. Data can be shown in the context of the tools that accessed or may access this data. We think that such a *flow-based representation* is an ideal instrument to support the kind of user interaction that we like to offer in our design flow user interface. We call this type of user interaction *flow-based user interaction.*

A large number of systems have been presented that support some form of flow-based user interaction [Janni86, Casotto90, Bretschnei90, Hamer90, Liebisch92, Kupitz92, Rumsey92, Sutton93]. However, these systems differ in the way they combine the design flow configuration and the run-time information into a flow-based representation of the design state. In particular, we observe a conceptual difference in the role they attach to the description of the design process as configured in the design flow configuration. We explain this difference in more detail using the example design flow configuration and run-time information shown in figure 5.2.

With respect to this example, we note that, strictly speaking, we can only identify a particular instance of an activity, port or channel in a hierarchical design flow configuration by the full path of flow hierarchy relationships to the root flow graph (see section 4.3). However, in this chapter we suppose that each flow graph is instantiated only once, so that we can identify each instance of an activity, port or channel simply by its name. We introduce this simplification because it contributes to a more simple and understandable explanation of the concepts to be addressed in this chapter. So, in figure 5.2, we simply name activity instances *a*, *b*, *c* etc. and port instances *p* 1, *p* 2, *p* 3 etc. and we will talk about activities and ports instead of activity instances and port instances. Due to this simplification, it suffices to describe the run-time information for this example by a table that has been obtained via the OTO-D query:

GET DataAccess ITS DataEntity, ActivityRun, ActivityRun ITS Activity, Port



DataAccess	DataEntity	ActivityRun	Activity	Port
da1	de1	ar1	a	p2
da2	de1	ar2	a	p1
da3	de2	ar2	a	p2
da4	de2	ar3	a	p1
da5	de3	ar3	a	p2
da6	de4	ar4	c	p4
da7	de1	ar5	b	p3
da8	de3	ar6	d	p5
da9	de4	ar6	d	p6
da10	de5	ar6	d	p7
da11	de2	ar7	d	p5
da12	de4	ar7	d	p6
da13	de6	ar7	d	p7
da14	de3	ar8	b	p3
da15	de5	ar9	e	p8

Figure 5.2. Example design flow configuration and run-time information.

If we classify the different flow-based representation techniques according to the role they attach to the design flow configuration, we obtain a spectrum of different representation techniques. At one side of this spectrum we have the approaches that do not use the design flow configuration at all and visualize the run-time information in a graph-based representation to the end-user. In our design flow system this technique would lead to an acyclic bipartite graph of activity runs and data entities. For each activity run the activity is shown with the data entities involved in that run on its input and output ports.

Figure 5.3 shows this representation technique for the example of figure 5.2. Note that no data is shown on the input port of the activity showing activity run *ar* 1, since data entity *de* 1 has been created from scratch. Since the graph grows upon each activity run performed, a user interface that uses this representation technique needs to offer pruning techniques to cope with large graphs. A design system that uses this type of representations for user interaction can be found in [Casotto90]. In the approach of Casotto, the choice for this representation technique was straightforward since the special character of this system (it traces the design process) makes that it has to do without a design flow configuration. We borrow the terminology used by Casotto and name such a graph of data entities and activity runs a *design trace*.

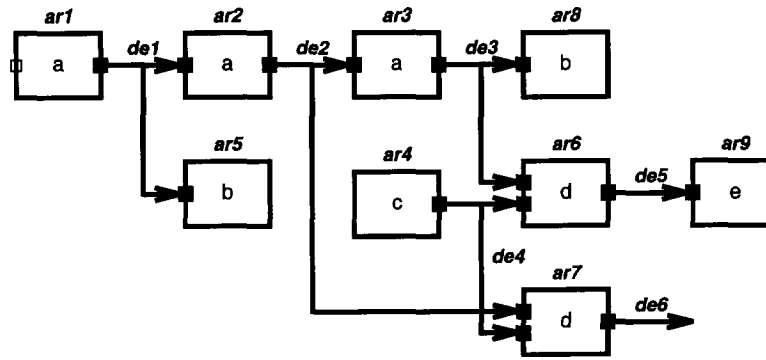


Figure 5.3. A design trace.

A major disadvantage of using a design trace for user interaction is that it visualizes the design actions that *have been* performed rather than visualizing the design actions that *can be* performed. For instance, in figure 5.3 it cannot be seen that activity *b* can be invoked on data entity *de* 2. The reason for this is that a design trace simply shows the history of the design process, without utilizing the information configured in the design flow. It neglects the description of the design process as described by the design flow configuration.

At the other side of the spectrum of flow-based representation techniques, we have the approaches that take the description of the design process as configured in the design flow configuration as a starting point. In our design flow system this leads to a representation that shows the activity runs and data entities in the context of the configured design flow. Since in such a representation each flow primitive (i.e. each port, channel or flow graph) is "colored" with data entities or activity runs, we call such a representation a *colored flow*. An advantage of the use of colored flows is that each design state is always presented in the same static view on the design process and that a designer can see at a glance which activities *have been* invoked as well as which activities *can be* invoked. Figure 5.4 shows a colored flow for the design flow configuration and run-time information of figure 5.2. It can be seen quite easily that activity *b* can be invoked on data entity *de* 2. An approach that uses this representation technique, where design flows are expressed as predicate transition nets, can be found in [Kupitz92].

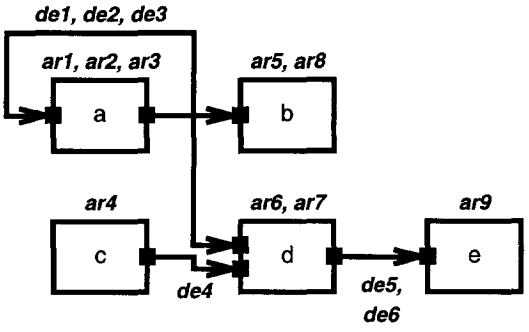


Figure 5.4. A colored flow.

A problem with the representation as shown in figure 5.4 is that it does not clearly show the derivation relationships between the data entities. For instance, it is unclear whether *de* 5 has been derived from *de* 1, *de* 2 or *de* 3. A solution to this problem is to limit the number of data entities and activity runs on the flow primitives to one at most, and to allow only directly related data entities (according to definition 4.3) to be shown on the ports of an activity. We call this type of constraints *flow coloring constraints*. Figure 5.5 shows a colored flow that satisfies the flow coloring constraints mentioned above, for the design flow configuration and run-time information of figure 5.2. Colored flow primitives are shown thick and uncolored flow primitives are shown thin. Note that this picture contains information on the design state, like the fact that data entity *de* 6 has been derived from data entities *de* 2 and *de* 4, as well as information on the executability of activities, like the fact that activity *b* can be executed on data entity *de* 2. Two approaches that use this type of colored flows for user interaction are [Janni86] and [Hamer90].

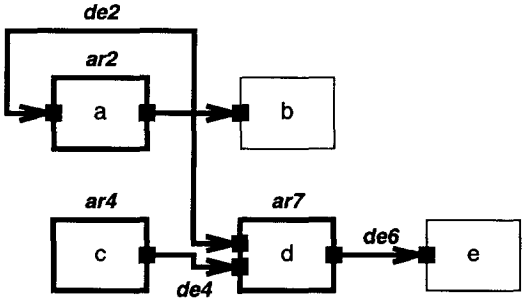


Figure 5.5. A colored flow satisfying some flow coloring constraints.

Besides a design trace and a colored flow, a variety of other flow-based representations can be found. For instance, figure 5.6 shows a flow-based representation put together from part of the colored flow of figure 5.5 and part of the design trace of figure 5.3. A design flow system that uses such representations (called *task graphs*) can be found in [Sutton93].

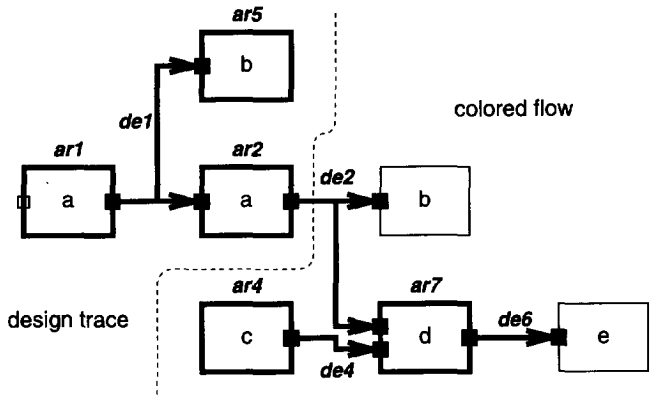


Figure 5.6. A combination of a design trace and a colored flow.

Now that we presented a number of different flow-based representation techniques, we can choose one of them for the design flow user interface. We think that a design trace or a combination of a design trace and a colored flow are less suited as the basic representation technique of a design flow user interface, since they neglect the important description of the design process as configured in the design flow configuration. It is our experience that the consequent use of one stable graphical description of the design process, such as a configured design flow, contributes to a natural and flexible interaction of the design system with the end-users. This makes the choice for a colored flow straightforward.

With respect to the different types of colored flows, we think that the unambiguous presentation of (part of) the design state should have high priority. Since the presentation of multiple data entities or activity runs on a single flow primitive makes it impossible for designers to view the important derivation relationships, we decide to limit the number of data entities or activity runs to be shown to one per flow primitive at most. In addition, it will appear to be necessary to formulate a number of other constraints to which a colored flow must adhere. From now on, we use the term colored flow only for representations that do satisfy these constraints.

The main problem with the use of colored flows as defined above is that a colored flow shows only part of the design state. However, we think that the advantage of an unambiguous representation of the design state counterbalances the disadvantage that the amount of information that can be shown simultaneously is limited. In fact, such a limited view on the design state may be an advantage. After all, it is generally recognized that an overkill of information can seriously hamper the success of a user interface [Sommervill85]. Also, because the state of the design is always presented in the same graphical description of the design process, the understanding of the information displayed may be more effective. It is our experience that the amount of information that can be displayed in a colored flow does satisfy the information needs of the designers.

Figure 5.7 shows the picture of figure 5.1, refined for the use of colored flows. A *flow coloring* algorithm builds a colored flow from the design flow configuration and the run-time information in the meta data. Each colored flow must satisfy the flow coloring *constraints*. To derive a colored flow that allies as much as possible to the interest of the designer, the flow coloring algorithm uses a number of *heuristics* and possibly some *hints* from the designer.

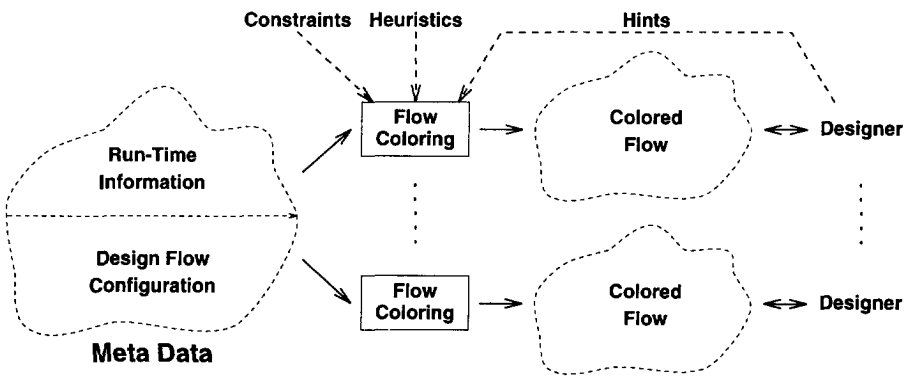


Figure 5.7. Flow-based user interaction based on colored flows.

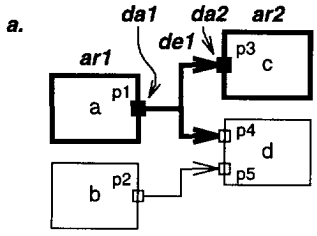
5.3 Flow Coloring

In the previous section we informally described how flow primitives can be colored with run-time information to display part of the design state and to visualize the executability of activities. In section 5.3.1 we explain this in more detail. The fact that a colored flow is to be used for design state viewing as well as activity invocation is important to be taken into account when defining colored flows. This objective, together with the requirement that we aim at a representation that shows the state of design in an unambiguous way, determines the set of *flow coloring constraints* that must be satisfied by each colored flow. For instance, we do not want to show unrelated data entities as if they were related. These flow coloring constraints are described in section 5.3.2. In section 5.3.3 we present the *switch* construct, which improves the use of colored flows for design state viewing and activity invocation. Since only part of the design state can be presented in a colored flow, it is essential that the end-user is supplied with operations to browse and change colored flows. In section 5.3.4 we present a number of *browsing operations* for colored flows that can be used to retrieve information on the design state related to the information that is shown in the colored flow. In section 5.3.5 we describe a number of *flow coloring operations* that can be used to transform one colored flow into another. On top of these browsing and flow coloring operations additional functionality may be built, like the automatic derivation of a *maximal colored flow*. This is addressed in section 5.3.6. Finally, in section 5.3.7 we describe how the presented flow coloring mechanisms can be extended to support the coloring of compound flow graphs.

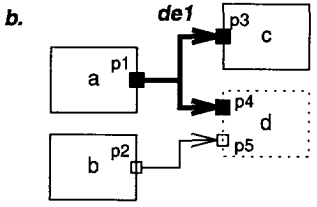
5.3.1 Coloring Flow Primitives

In a colored flow, flow primitives are *colored* with run-time information. We like to define the coloring of flow primitives in such a way that the resulting colored flows can be used for design state presentation as well as for activity invocation. However, it appears that these are competing requirements. For visualizing the design state, it would be best to color an activity with a run of this activity, to color a port with a data access that has been performed on a data entity via this port and to color a channel with a data entity. In this way, a colored flow shows which data entities have been accessed via which ports in the context of which activity run. Figure 5.8.a. shows this method of coloring flow primitives. Colored activities and channels are shown thick and colored ports are drawn solid. Note that port $p4$ is not colored, since no data access has been performed (yet) on data entity $de1$ via this port. For visualizing the executability of activities on data entities, on the other hand, it would be better to color an activity with an 'executability' flag, to color a port with a data entity on which an activity may be

executed and to color a channel with a data entity. An example of this way of coloring flow primitives for the same run-time information as in figure 5.8.a is shown in figure 5.8.b. Colored channels are shown thick, colored ports are drawn solid and activities that are not executable are dotted. Note that port $p4$ is colored, since data entity $de1$ can be accessed via this port.



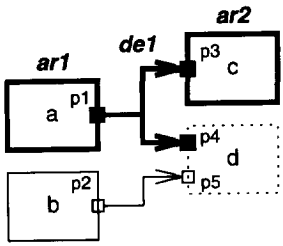
DataAccess	DataEntity	ActivityRun	Activity	Port
da1	de1	ar1	a	p1
da2	de1	ar2	c	p3



DataAccess	DataEntity	ActivityRun	Activity	Port
da1	de1	ar1	a	p1
da2	de1	ar2	c	p3

Figure 5.8. Two methods of flow coloring.

Because colored flows are to be used for design state viewing as well as for activity invocation, we like to merge both methods into one method of coloring flow primitives. Since the simultaneous coloring of an activity with an activity run and a port with a data entity can be used to describe a data access to the data entity that colors this port, there is no need to color ports with data accesses explicitly and we choose to color ports and channels with data entities and activities with activity runs. In addition, for activities that are not colored with an activity run, we show the executability with respect to the data entities that color its input ports. For the situation of figure 5.8, this leads to a coloring of the flow primitives as shown in figure 5.9. Colored activities and channels are shown thick, colored ports are drawn solid and activities that are not colored with an activity run are drawn with regular lines (executable) or dotted (not executable).



DataAccess	DataEntity	ActivityRun	Activity	Port
da1	de1	ar1	a	p1
da2	de1	ar2	c	p3

Figure 5.9. A mixture of the two methods of flow coloring.

Using this method of coloring flow primitives, a colored flow can be described by 3 coloring functions and one executability function[1]:

- a *port coloring* function: $color_{port}$: PORT $\rightarrow$ DATAENTITY.
  This function assigns a data entity to a port.
- a *channel coloring* function: $color_{channel}$: CHANNEL $\rightarrow$ DATAENTITY.
  This function assigns a data entity to a channel.
- an *activity coloring* function: $color_{activity}$: ACTIVITY $\rightarrow$ ACTIVITYRUN.
  This function assigns an activity run to an activity.
- an *executability* function: $executability_{activity}$: ACTIVITY $\rightarrow$ {*true*, *false*}
  This function determines the executability of an activity with respect to the data entities that color its input ports.

We write $color(x)$ for $color_{port}(x)$, $color_{channel}(x)$ or $color_{activity}(x)$ if the type of x is clear and no confusion can arise. Since some ports, channels and activities may be uncolored in a colored flow, the port, channel and activity coloring functions may be partial. With $color(x)\downarrow$ we mean that flow primitive x is colored and with $color(x)\uparrow$ we mean that flow primitive x is not colored. The executability function is total.

1. Note that the sets such as PORT, DATAENTITY etc. denote the set of instances of the corresponding object type in the information model, as explained in section 4.2.1.

5.3.2 Flow Coloring Constraints

In order to let a coloring of flow primitives describe an understandable and meaningful representation suitable for design state viewing as well as for activity invocation, it must satisfy a number of *flow coloring constraints* (FCC's). These constraints specify dependencies between the coloring functions and the executability function. First of all, we specify under which conditions an activity, port or channel may be colored. For the coloring of activities it is straightforward to require that an activity may be colored with an activity run of that particular activity only. This leads to the first flow coloring constraint:

$$\forall_{a \in ACTIVITY, color(a)\downarrow} [Activity(color(a)) = a] \qquad \text{(FCC1)}$$

To specify the conditions for the coloring of ports and channels we recall from the previous section that, in order to let a colored flow show via which ports a data entity has been *produced* as well as via which port it can be *accessed*, data entities should 'flow' through channels to connected ports. For instance, in figure 5.9 not only port $p1$ is colored with data entity $de1$, the channel linked to this port and all connected ports ($p3$ and $p4$) are colored with this data entity too, to indicate that $de1$ can be accessed via these ports. On the other hand, not all data entities may color a port or channel. For instance, port $p5$ in figure 5.9 may not be colored with data entity $de1$ since the connectivity of the activities denotes that $de1$ can not be accessed via this port. Summarizing, a port may be colored with a data entity, only if this port is connected to the port via which this data entity has been produced, which we call the *production port* of a data entity[2]:

$$\forall_{p \in \text{PORT}, color(p)\downarrow} [conn(prod(color(p)), p)] \qquad \text{(FCC2)}$$

where $prod(de)$ is the port for which there is a data access $da \in$ DATAACCESS with $Port(da) = p$, $DataEntity(da) = de$ and $AccessMode(p) = output$.

The same holds for channels. Since FCC2 already guarantees that a colored port is connected to the production port of the data entity that colors this port, we only have to require that a colored channel is adjacent to a port that is colored with the same data entity:

2. Since we made the assumption that each flow graph is instantiated only once (see section 5.2), it is correct to speak about *the* production port of a data entity. Without this simplification it would be possible that a data entity is produced by multiple instances of the same port. However, this does not make a conceptual difference for the flow coloring principles.

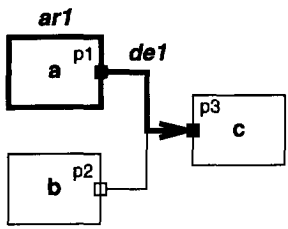
$$\forall_{ch \in \text{CHANNEL}, color(ch)\downarrow} \left[\exists_{p \in \text{PORT}, color(p)=color(ch)} [adj(p, ch)] \right] \quad \text{(FCC3)}$$

Above, we defined under what conditions a port or channel may be colored with a data entity. The 'flow' of data entities through channels and ports in a colored flow implies that adjacent ports and channels must be colored with the same data entity:

$$\forall_{p \in \text{PORT}, color(p)\downarrow} \left[\forall_{ch \in \text{CHANNEL}, adj(p,ch)} [color(ch)=color(p)] \right] \quad \text{(FCC4)}$$

$$\forall_{ch \in \text{CHANNEL}, color(ch)\downarrow} \left[\forall_{p \in \text{PORT}, adj(ch,p)} [color(p)=color(ch)] \right] \quad \text{(FCC5)}$$

A side-effect of FCC4 and FCC5 is that the coloring of one port may prevent other ports from being colored. An example of such a situation is shown in figure 5.10 (because we do not assign data accesses to flow primitives, from now, we do not show the column for data accesses in the table with the run-time information). The coloring of port $p1$ makes that port $p2$ must stay uncolored, because otherwise port $p3$ would have to be colored with two data entities, which we forbid for reasons of clarity.



DataEntity	ActivityRun	Activity	Port
de1	ar1	a	p1
de2	ar2	b	p2

Figure 5.10. The coloring of port $p1$ implies the coloring of port $p3$ and prevents the coloring of port $p2$.

With respect to the executability function, we define an activity *executable* if and only if all its required input ports are colored:

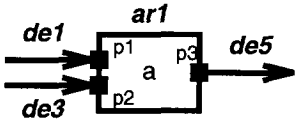
$$executable(x) = true \text{ iff } \forall_{p \in RIP(x)} [color(p)\downarrow] \quad \text{(FCC6)}$$

where $RIP(x)$ is the set of *R*equired *I*nput *P*orts of activity x,

$$RIP(x) = \{p \in PORT \mid \begin{bmatrix} Activity(p) = x, \\ AccessMode(p) = input, \\ Multiplicity(p) = required \end{bmatrix} \}$$

In the previous section we decided to let the simultaneous coloring of a port with a data entity and an activity with an activity run denote a data access performed via that port. Therefore, to visualize the design state in an unambiguous way, we do

not want to admit a colored flow in which an activity is colored with an activity run and one of its ports is colored with a data entity that has not been involved in this activity run. Such a situation, which we call an *inconsistency*, may cause an end-user to draw incorrect conclusions about the involvement of data entities in an activity run and about the relatedness (see definition 4.3) of data entities. For instance, consider the colored flow shown in figure 5.11. The coloring of activity a with activity run $ar1$ and its ports with data entities $de1$, $de3$ and $de5$ suggests that all these data entities are related through activity run $ar1$. However, $de5$ has not been derived from $de1$ and $de3$ in activity run $ar1$, as can be seen from the table.



DataEntity	ActivityRun	Activity	Port
de1	ar1	a	p1
de3	ar1	a	p2
de4	ar1	a	p3
de2	ar2	a	p1
de3	ar2	a	p2
de5	ar2	a	p3

Figure 5.11. A flow coloring with an inconsistency.

The prevention of inconsistencies leads to the flow coloring constraint that for each colored port of a colored activity, the data entity that colors the port is involved in the activity run that colors the activity:

$$\underset{a \in ACTIVITY}{\forall} [inc(a) = false] \qquad \text{(FCC7)}$$

where $inc(a)$ holds if there is an inconsistency on activity a:

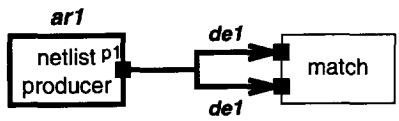
$$inc(a) \text{ iff } color(a)\downarrow \text{ and } \underset{p \in PORT, Activity(p)=a, color(p)\downarrow}{\exists} [color(a) \notin AR_p]$$

and where AR_p is the set of *A*ctivity *R*uns in which the data entity on port p is involved,

$$AR_p = \{ar \in ACTIVITYRUN \mid \underset{da \in DATAACCESS}{\exists} \begin{bmatrix} ActivityRun(da) = ar, \\ Port(da) = p, \\ DataEntity(da) = color(p) \end{bmatrix}\}$$

5.3.3 Switches

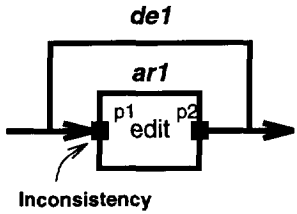
According to flow coloring constraints 4 and 5 data always 'flows' through channels to adjacent ports. This constraint was introduced to show which activities may be executed on which data entities. However, there are situations where this is particularly uncomfortable. For instance, consider the design flow in figure 5.12. An activity *match* is used to analyze the differences between two netlist descriptions. If this design flow is colored with a data entity produced by the netlist production activity, this data entity will color both input ports of the activity *match*. This implies that *match* can only be invoked on identical data entities. However, the access rule in the underlying design flow model (see definition 4.2) specifies that match may certainly execute on different data entities.



DataEntity	ActivityRun	Activity	Port
de1	ar1	netlist producer	p1
de2	ar2	netlist producer	p1

Figure 5.12. The activity *match* can only compare one and the same netlist.

Another situation in which the 'flow' of data entities through channels is particularly uncomfortable, can be found in design flows with activities that may consume their own output data, like editors. For instance, in figure 5.13 flow coloring constraint 7 prohibits the simultaneous coloring of the editor activity and its input and output port, since there is no editor run in which a data entity has been produced as well as consumed.



DataEntity	ActivityRun	Activity	Port
de1	ar1	edit	p2
de1	ar2	edit	p1
de2	ar2	edit	p2

Figure 5.13. The *editor* and its ports cannot be colored simultaneously.

To invoke the match activity from a colored flow on different netlists, and to make it possible to show an editor run together with the data involved in this editor run in one colored flow, we need a way to prevent data entities from flowing through channels. We like to temporarily "switch off" the flow of data through channels to adjacent ports. We introduce a *switch* for this. A switch is a special type of flow graph with one input port, multiple output ports and one channel that links the

input port to one of the output ports[3]. The connectivity of the channel to one of the output ports can be changed to influence the flow of data entities in a colored flow. In a colored flow, a switch is represented by drawing its channel only. Figure 5.14 shows the configuration of a switch with two output ports and its representation in a colored flow. By changing the connectivity of the channel of the switch, the flow of data entities can be redirected. In this example one of the output ports is left unlinked, making it a kind of tap with two states: "open" and "closed".

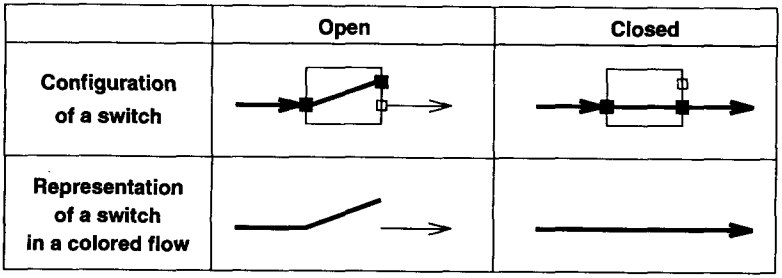
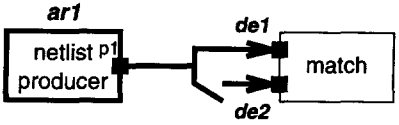


Figure 5.14. A *switch* can be used to redirect the flow of data entities.

Figure 5.15 shows the use of a switch to color the inputs of the match activity with two different data entities

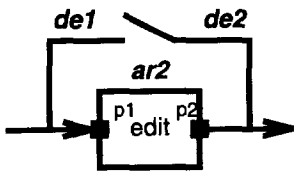


DataEntity	ActivityRun	Activity	Port
de1	ar1	netlist producer	p1
de2	ar2	netlist producer	p1

Figure 5.15. The switch makes it possible to invoke *match* on different netlists.

The switch in figure 5.16 makes that both *de* 1 and *de* 2 and activity run *ar* 2 can be shown in one single colored flow.

3. It is possible to extend the switch concept to allow switches with multiple input and multiple output ports. This may be useful to select the data of one of multiple producer activities to flow to one of multiple consumer activities. However, in this thesis we limit ourselves to 1-to-n switches, which offer enough modeling power to solve the problems mentioned in this section.



DataEntity	ActivityRun	Activity	Port
de1	ar1	edit	p2
de1	ar2	edit	p1
de2	ar2	edit	p2

Figure 5.16. The switch in the editor loop enables the simultaneous presentation of data entities *de* 1 and *de* 2 and activity run *ar* 2.

The use of switches makes that some of the concepts presented so far must be refined. First of all, for a complete description of a colored flow, we have to describe the state of the switches as well. Since the state of a switch can be described by the output port to which its channel is connected, the description of a colored flow should be extended with a function that assigns a port to a switch:

- a *switch state* function: $state_{switch}$: SWITCH $\rightarrow$ PORT.
  SWITCH $\subseteq$ FLOWGRAPH.

Second, to allow an output port p of a switch s with $state_{switch}(s) \neq p$ to be colored, we have to refine flow coloring constraint 2. For instance, data entity *de* 1 in figure 5.16 colors an output port of a switch, while this output port is not connected to the production port of *de* 1. Therefore, the relationship *conn* in flow coloring constraint 2 should be replaced by the relationship $conn_{ignore\ switch}$, which is defined as follows (compare it to the definition of *conn* in section 4.3):

$$conn_{ignore\ switch}(p_1,p_2) \text{ iff } p_2 \in \Gamma^*_{adj_{ignore\ switch}}(p_1)$$

where

$$adj_{ignore\ switch}(p_1,p_2) \text{ iff } \begin{cases} adj(p_1,p_2) \\ \quad \text{or} \\ \exists\limits_{s \in SWITCH} \begin{bmatrix} AccessMode(p_1) = input, \\ AccessMode(p_2) = output, \\ Activity(p_1) = Activity(p_2) = s \end{bmatrix} \end{cases}$$

An example situation where $adj_{ignore\ switch}(p_1,p_2)$ holds and $adj(p_1,p_2)$ does not hold is shown in figure 5.17. Note that $s \in$ SWITCH.

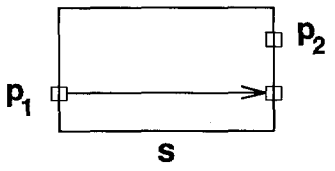


Figure 5.17. Situation in which holds $adj_{ignore\ switch}(p_1, p_2)$.

Summarizing, a colored flow can be specified completely by the flow coloring functions $color_{port}$, $color_{channel}$, $color_{activity}$, $executability_{activity}$ and $state_{switch}$. However, since flow coloring constraints 2 and 4 make that the activity executability and the channel coloring function can be calculated if the port coloring function is known, a colored flow is already fully specified by its port coloring function, its activity coloring function and the state of its switches. Therefore, in the next sections we limit the description of a colored flow to these three functions.

5.3.4 Browsing a Colored Flow

Since a colored flow as defined in the previous sections displays only part of the design state, it is useful to offer a number of so-called *browsing operations* that may be used by the designers to inspect those parts of the design state that are related to the information shown in a colored flow. In this section we describe two of such browsing operations Each browsing operation actually corresponds to a complex query about the state of design. We note that other browsing operations than the ones presented here may be offered in a design flow user interface as well. The browsing operations described here are those operations that we have found to be useful in our prototype implementations.

The first browsing operation is defined with respect to a specific port and enables a designer to inspect the derivation relationships between different data entities or to retrieve all data entities that may color a particular port. If a port of an activity is selected, it returns all data entities that may color this port and that are related to the data entities that color the other ports of the activity. If all other ports of this activity are uncolored, a list of all data entities that may color the selected port is returned. We define the operation *browse*(p), $p \in$ PORT in c-alike pseudo code. The set CP_p denotes the set of *C*olored *P*orts of *Activity*(p) not equal to p and DE_p denotes the set of *D*ata *E*ntities produced or accessed via port p.

Browse (p)

$CP_p = \{p' \in \text{PORT} \mid p' \neq p,\ Activity(p') = Activity(p),\ color(p')\downarrow\};$
if $(CP_p = \emptyset)$ {
  return $(\{de \in \text{DATAENTITY} \mid conn_{ignore\ switch}(prod(de), p)\});$
}
else {
  $DE_p = \{de \in \text{DATAENTITY} \mid \underset{da \in \text{DATAACCESS}}{\exists} [Port(da) = p,\ DataEntity(da) = de]\};$
  return $(\{de \in DE_p \mid \underset{p' \in CP_p}{\forall} [rel(color(p'), de)]\});$
}

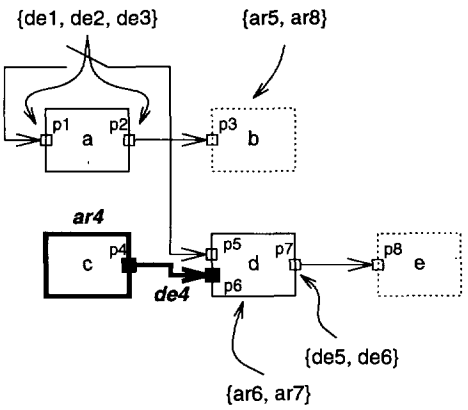
The second browsing operation is defined with respect to a specific activity and enables a designer to inspect the activity runs in which data entities have been involved or to retrieve all activity runs that may color an activity. If an activity is selected, this operation returns a list of activity runs that may color this activity, and in which the data entities that color the ports of the activity have been involved. If all ports of the activity are uncolored, all activity runs of this activity are returned. In the definition of the operation $browse(a), a \in \text{ACTIVITY}$, below, AR_a denotes the set of *A*ctivity *R*uns of activity a and CP_a denotes the set of *C*olored *P*orts of a:

Browse (a)

$AR_a = \{ar \in \text{ACTRUN} \mid Activity(ar) = a\};$
$CP_a = \{p \in \text{PORT} \mid Activity(p) = a,\ color(p)\downarrow\};$

$$return\ (\{ar \in AR_a \mid \underset{p \in CP_a}{\forall} \underset{da \in \text{DATAACCESS}}{\exists} \begin{bmatrix} Port(da) = p, \\ DataEntity(da) = color(p), \\ ActRun(da) = ar \end{bmatrix}\});$$

We give an example of the use of the browsing operations in figure 5.18. It shows a colored flow for the design flow configuration and run-time information of figure 5.2. Since activity a has no colored ports, the selection of port $p1$ or $p2$ results in the set of data entities that may color these ports. For both ports, this is the set $\{de1, de2, de3\}$. Upon the selection of activity b the user interface gives all runs of activity b: $\{ar5, ar8\}$. Since port $p6$ of activity d is colored with data entity $de4$ the selection of activity d results in all runs of this activity in which $de4$ is involved via port $p6$: $\{ar6, ar7\}$ and the selection of port $p7$ results in all data entities that were derived from $de4$: $\{de5, de6\}$.



DataEntity	ActivityRun	Activity	Port
de1	ar1	a	p2
de1	ar2	a	p1
de2	ar2	a	p2
de2	ar3	a	p1
de3	ar3	a	p2
de4	ar4	c	p4
de1	ar5	b	p3
de3	ar6	d	p5
de4	ar6	d	p6
de5	ar6	d	p7
de2	ar7	d	p5
de4	ar7	d	p6
de6	ar7	d	p7
de3	ar8	b	p3
de5	ar9	e	p8

Figure 5.18. The use of browsing operations in a colored flow and their results.

5.3.5 Flow Coloring Operations

Due to the objective to let a colored flow represent the design state in an unambiguous way, a colored flow can only show part of the total design state as administered in the database of run-time information. The browsing operations presented in the previous section already enable the retrieval of additional information from a colored flow. However, to perform multiple design actions from one and the same flow-based representation, designers should also have the ability to direct the design flow user interface to show a colored flow that displays the information they are interested in. They must be able to indicate in which data entities or activity runs they are interested and the design flow user interface should respond appropriately. Therefore, besides a clear definition of what we consider to be a valid colored flow, we also have to define mechanisms to transform one colored flow into another colored flow. For this, we develop a transformation system for colored flows. We describe a *minimal* set of *flow coloring operations* that is *sound* and *complete* with respect to colored flows.

A set of flow coloring operations is *sound* if, starting from a colored flow, any sequence of flow coloring operations results in another colored flow.

A set of flow coloring operations is *complete* if, starting from a colored flow, any other colored flow can be derived using a sequence of flow coloring operations.

A set of flow coloring operations is *minimal* if any subset of this set of flow coloring operations is not complete.

To keep the transformation system simple, we use flow coloring operations that describe the smallest changes that can be made to a colored flow, taking into account that it should yield another colored flow. More powerful user interface operations can be expressed in terms of these basic flow coloring operations.

The first flow coloring operation is the *clear* operation. This operation can be applied to a port p or an activity a and transforms a colored flow into another colored flow in which this port or activity is uncolored. To let the resulting colored flow satisfy flow coloring constraints 2 and 4, the clear operation for ports should not only uncolor the port p but also all ports that can be reached by applying the relationship *adj* any times in either direction. This set of ports corresponds to the reflexive and transitive closure of the port to be cleared, with respect to the relationship *adj'* written as $\Gamma^*_{adj'}(p)$, where $adj'(p_1,p_2)$ holds if and only if $adj(p_1,p_2)$ or $adj(p_2,p_1)$ holds. The clear operation for an activity is much simpler. A colored flow in which one colored activity is uncolored does always satisfy the flow coloring constraints. The clear operations are defined as follows:

Clear(p)
for all $p' \in \Gamma^*_{adj'}(p)$ do {
  $color(p') = \uparrow$;
}

Clear(a)
$color(a) = \uparrow$;

The second flow coloring operation is the *assignment* of a data entity *de* to a specific port p or an activity run *ar* to a specific activity a. One way of selecting a data entity or activity run for assignment is to select it from a set that is retrieved via one of the browsing operations described in the previous section. In design systems that support multiple browsers for visualizing the design state from different viewpoints (see for instance [Gedye88] and [Bingley90]) and that use the well-known *drag & drop* mechanism for passing information between browsers, this mechanism can be used for selecting data entities or activity runs from outside the flow-based user interface. The assignment operation of a data entity or an activity run adjusts a colored flow in such a way that it shows the data entity or the activity run in the new colored flow. To let the assignment of a data entity to a port satisfy flow coloring constraints 2 and 4, it is necessary to perform a clear operation before coloring the rest of the flow. Also, to let the resulting colored flow satisfy flow coloring constraint 7, it must be checked whether inconsistencies arise on activities, and if so these activities must be cleared. The same holds for

the assignment of an activity run to an activity. The resulting colored flow must be checked for inconsistencies and if one is detected, the appropriate activities should be cleared.

Assign(*de,p*)

if ($conn_{ignore\ switch}(prod(de),p)$) {
  $clear(p)$;
  for all $p' \in \Gamma^{*}_{adj'}(p)$, $conn_{ignore\ switch}(prod(de),p')$ do {
    $color(p') = de$;
  }
  for all $a \in$ ACTIVITY, inc(a) do {
    $color(a) = \uparrow$;
  }
}

Assign(*ar,a*)

if ($Activity(ar) = a$) {
  $color(a) = ar$;
  for all $p \in$ PORT, $Activity(p)=a$, $color(p)\downarrow$, $ar \notin AR_p$ {
    $clear(p)$;
  }
}

The last flow coloring operation is the manipulation of the state of a switch. Since the state of a switch is determined by the port to which its channel is connected, we need a flow coloring operation to set the state of a switch s to a specific port p. To satisfy flow coloring constraints 2 and 4, the coloring of port p should equal the coloring of the input port of the switch. Therefore, if the input port is uncolored, port p should be cleared and otherwise the data entity of the input port should be assigned to port p.

Switch(s,p)

```
if (color(state_switch(s))↑) {
   state_switch(s)=p;
   clear(p);
}
else {
   de = color(state_switch(s));
   state_switch(s)=p;
   assign(de,p);
}
```

As stated at the beginning of this section, we aim at a set of flow coloring operations that is *complete* with respect to colored flows. The flow coloring operations presented have this property, since any colored flow can be transformed into any other colored flow using a finite number of clear operations to arrive at the empty colored flow, a finite number of switch operations to put all switches in the right state and a finite number of assign operations to color flow primitives with data entities and activity runs.

We also stated at the beginning of this section that we aim at a set of flow coloring operations that is *minimal* with respect to colored flows. This holds for the flow coloring operations presented, since if any of these is left out, the transformation system is no longer complete. After all, without clear operations it is impossible to uncolor ports and activities, without assignment operations it is impossible to color them and without the switch operation it is impossible to change the state of a switch.

The flow coloring operations *clear*(p), *clear*(a), *assign*(de,p), *assign*(ar,a) and *switch*(s,p) offer a basic level of flow coloring functionality on top of which additional functions can be built by the user interface. If all other user interface operations use these basic flow coloring operations, the soundness of the user interface with respect to colored flows is guaranteed, i.e. each user interface operation results in another valid colored flow.

5.3.6 The Derivation of a Maximal Colored Flow

In the previous sections we presented a number of browsing and flow coloring operations that define the basic functionality of a flow-based user interface. Using these operations designers can browse through all parts of the design state. However, in some situations it is quite obvious in what part of the design state a designer is most interested and we would like the design flow user interface to focus on those parts automatically. In this section we describe such a user interface function: the automatic derivation of a *maximal colored flow*. A maximal colored flow is a colored flow for which holds that all data entities and activity runs that color the flow primitives are related and in which no uncolored flow primitives exist that could be colored with related data entities or activities without violating any of the flow coloring constraints.

Given a design flow configuration and a specific design state, there may be many different maximal colored flows, each displaying a different part of the design state. To let a designer indicate in which part of the design state he is particularly interested, a designer should specify an initial data entity. We call this data entity the *user-assigned* data entity. The global idea for deriving a maximal colored flow starting from a user-assigned data entity, is the repeated assignment of activity runs and data entities related to the user-assigned data entity. In each pass of the derivation process the run-time information is inspected to see whether uncolored ports and uncolored activities can be colored with related data and activity runs. This process continues until no more flow primitives can be colored without violating any of the flow coloring constraints.

An example of the derivation of a maximal colored flow for the design flow configuration and run-time information of figure 5.2 is shown in figure 5.19. The colored flow in figure 5.19.a shows a situation where the user has assigned data entity $de1$ to the input port of activity a. In the first pass of the derivation process activity run $ar2$ is assigned to activity a since $de1$ has been involved in this run via port $p1$ and data entity $de2$ is assigned to port $p2$. The resulting colored flow is shown in figure 5.19.b. In the second pass activity run $ar7$ is assigned to activity d and $de4$ and $de6$ are assigned to port $p6$ and $p7$ respectively. Finally, the assignment of activity run $ar4$ to activity c results in a maximal colored flow as shown in figure 5.19.c. Activities b and e remain uncolored since coloring them with one of their runs would violate the flow coloring constraints.

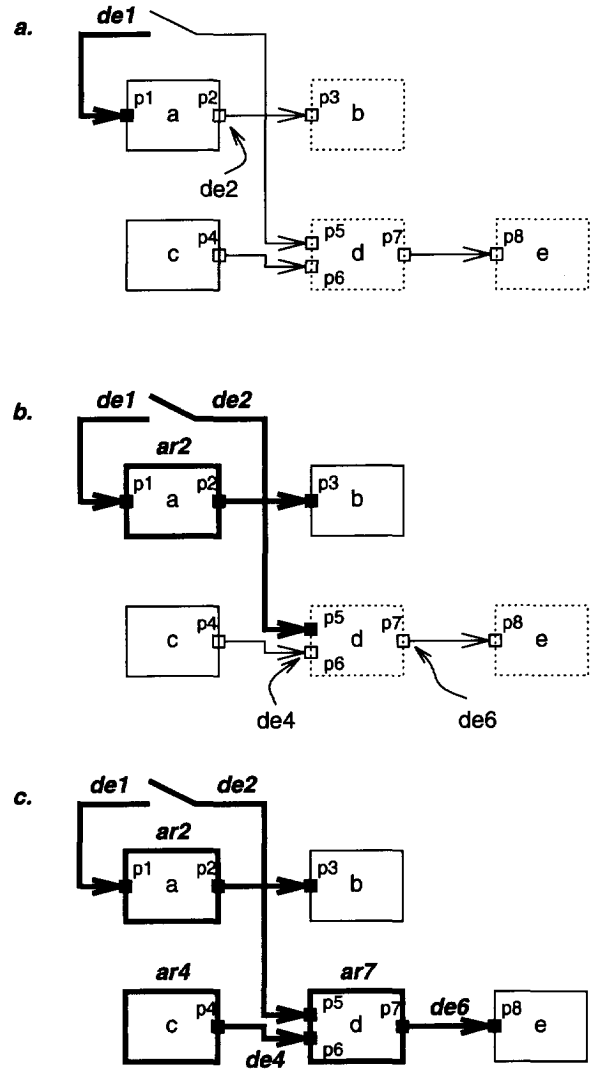


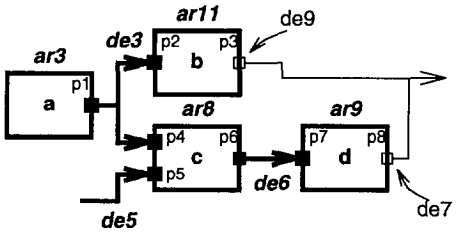
Figure 5.19. Deriving a maximal colored flow starting from the user-assigned data entity *de* 1.

In this example no choices had to be made about which data entities and activity runs to show, since in each pass there was only one data entity or activity run that was candidate for assignment. However, it is possible that multiple related data entities or activity runs are found for a single flow primitive. In this case, a selection must be made which of them to assign. For instance, if in the example of figure 5.19 *de* 2 would be involved in multiple runs of activity *d*, the derivation

process would have to make a choice which of these activity runs to assign to the colored flow. When making such selections, the end-goal is to derive a colored flow that allies as much as possible to the expectation and interest of the designer. We introduce a number of *heuristics* that we have found to be useful for making these selections.

The idea underlying the first heuristic is that designers are usually most interested in that part of the design state that reflects the most recently performed design actions. For example, if a new data entity has just been generated, a designer will typically like to see this data entity in the colored flow that is derived. Therefore, if choices have to be made in the derivation process we select data entities or activity runs based on their *recentness*. This heuristic implies that in the example of figure 5.19 where data entity $de2$ is involved in multiple activity runs of activity d, the most recent activity run would be chosen.

Although this heuristic gives a satisfying result in simple cases, it still leaves us with some problems in more complicated design flow configurations, especially those that contain a loop construct. An example of such a situation is shown in figure 5.20. Design data entity $de5$ has been assigned by the user and the user interface has assigned activity run $ar8$, data entities $de3$ and $de6$ and activity runs $ar3$, $ar11$ and $ar9$. In this situation $de9$ is candidate for assignment to port $p3$ and $de7$ to port $p8$. Since these ports are linked to the same channel and since a data entity that colors a port 'flows' into all channels that are linked to this port, only one of these data entities can be assigned. Using a selection on recentness, data entity $de9$ would be chosen (assuming that the numbers indicate their order of creation). However, intuitively we feel that it is more desirable to show data entity $de7$ than data entity $de9$, because $de7$ is directly derived from the user-assigned data entity $de5$, while $de9$ is only related to $de5$ because it is derived from another data entity ($de3$) that has been used as a fellow input in an activity run in which $de5$ has been accessed.



DataEntity	ActivityRun	Activity	Port
de3	ar3	a	p1
de3	ar8	c	p4
de5	ar8	c	p5
de6	ar8	c	p6
de6	ar9	d	p7
de7	ar9	d	p8
de3	ar11	b	p2
de9	ar11	b	p3

Figure 5.20. Either $de7$ or $de9$ can be shown in the colored flow.

This example leads to the observation that, in the view of the end-user, not all related data entities or activity runs are equally related. Some data entities are felt to be more related to the user-assigned data entity than others. In general, designers are more interested in a data entity or activity run if it is 'more related' to the user-assigned data entity. Therefore, we introduce a heuristic that gives preference to the 'most related' data entities or activity runs. We use the term *affinity* to denote the 'degree of relatedness' with respect to the user-assigned data entity. To let this heuristic work, it is necessary to calculate the affinity of each candidate data entity or activity run. For this, the derivation process administers *how* a data entity or activity run has been found in a *search graph*. This search graph has two types of nodes, nodes that describe an activity run and nodes that describe a data entity. The edges in the search graph describe via which activity runs and data entities other activity runs and data entities have been found. Since for each data entity and activity run there is only one node, and since each activity run can only be found via one or more data entities and each data entity via one or more activity runs, the search graph is a bipartite directed acyclic graph. Using this search graph, the affinity of each candidate activity run and data entity can be calculated from the affinity of the data entities and activity runs via which it was found and the type of search relationships that were used. We distinguish between three types of search relationships (we describe types of search relationships using family relationship names, see also figure 5.21):

1. We speak of an *ancestor* search relationship if a data entity is found on an input port of a colored activity or if an activity run is found for an activity for which an output port is colored.

2. We speak of a *descendent* search relationship if an activity run is found for an activity for which an input port is colored or if a data entity is found on an output port of a colored activity.

3. We speak of a *sibling* search relationship if a data entity is found by a descendent search relationship followed by an ancestor search relationship or if it is found via an ancestor search relationship followed by a descendent search relationship.

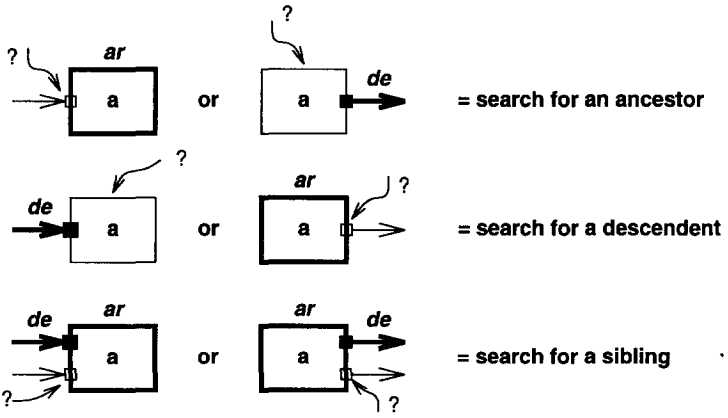


Figure 5.21. For the calculation of the affinity we make a distinction between three types of search relationships.

It is our experience that a satisfying result is obtained by giving the ancestor search relationship the highest priority, the descendent search relationship a lower priority and the sibling search relationship the lowest priority. In addition, to support the fact that the number of search passes needed to find a data entity or activity run plays a role, we let the affinity of a data entity or activity run decrease with the number of search passes that have been used to find it. It is possible to extend the determination of the affinity with other information maintained in a CAD framework. For instance, except for the derivation relationships between data entities, one could take other relationships among design data entities into consideration, like hierarchical and equivalence relationships.

As an example of the use of the affinity heuristic, we take a look at the example in figure 5.20 again. The search graph for this situation is shown in figure 5.22. Nodes for data entities are shown as circles and nodes for activity runs are shown as rectangles. Data entities or activity runs that are candidate for assignment are shown dashed. Each edge has a letter, indicating the search relationship used (*a*, *d* or *s*). Using the affinity heuristic we should decide to assign data entity *de* 7, because *de* 7 has been found via descendent search relationships only, and *de* 9 has been found via a sibling search relationship and two descendent search relationships.

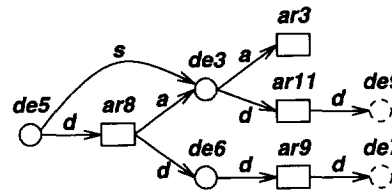
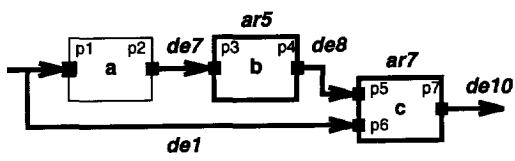


Figure 5.22. Search graph for figure 5.20.

Summarizing, we select data entities and activity runs to be assigned to a colored flow, based on the *affinity* with respect to the user-assigned data entity and the *recentness* of the data entity or activity run. We give the selection on the degree of relatedness a higher priority than selection on recentness. Only if multiple data entities with the same affinity are found we select on recentness.

The derivation process presented so far guarantees that no inconsistencies can arise, since it uses the sound set of flow coloring operations as presented in section 5.3.5. However, since the flow coloring operations remove data entities and activity runs from the colored flow if inconsistencies are about to arise, it may happen that the derivation process keeps assigning the same data entities and activity runs again and again. This is particularly likely to occur in design flows that contain a loop. The configuration in figure 5.23 gives an example of such a situation. It shows a colored flow, the run-time information and the search graph that is built during the derivation process. In the search graph can be seen that there are two candidates for assignment: activity run $ar3$ and activity run $ar4$. Since activity run $ar3$ has a higher affinity than activity run $ar4$ (it has been found using ancestor search relationships only), $ar3$ will be assigned to the colored flow.



DataEntity	ActivityRun	Activity	Port
de2	ar3	a	p1
de7	ar3	a	p2
de1	ar4	a	p1
de6	ar4	a	p2
de7	ar5	b	p3
de8	ar5	b	p4
de8	ar7	c	p5
de1	ar7	c	p6
de10	ar7	c	p7

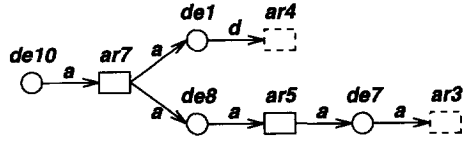
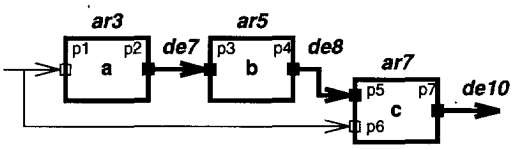


Figure 5.23. An almost completed derivation process.

This results in a situation as shown in figure 5.24. But in this situation the derivation process will continue and assign data entity $de1$ to port $p6$ (it has a higher affinity than $de2$ because there were less search passes needed to find it) resulting in the colored flow of figure 5.23 again.



DataEntity	ActivityRun	Activity	Port
de2	ar3	a	p1
de7	ar3	a	p2
de1	ar4	a	p1
de6	ar4	a	p2
de7	ar5	b	p3
de8	ar5	b	p4
de8	ar7	c	p5
de1	ar7	c	p6
de10	ar7	c	p7

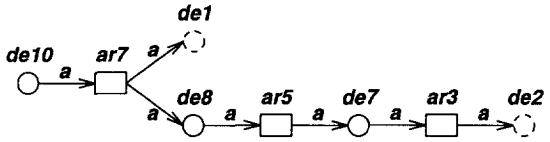


Figure 5.24. The assignment of $ar3$ removes data entity $de1$ from the colored flow.

Fortunately, the termination of the derivation process can be guaranteed in a simple way. Since all data entities and activity runs that have been found and assigned are administered in the search graph, a non-terminating situation can be detected by checking whether an activity run or data entity is removed that has previously been assigned. We conclude that the derivation process must stop when it is about to remove a data entity or activity run that is in the search graph.

5.3.7 Flow Coloring for Compound Flow Graphs

In the previous sections we focussed on the coloring of ports, channels and activities only. We did not address the coloring of compound flow graphs yet. However, as described in section 4.3, in a powerful design flow system design flows may be organized hierarchically and designers may operate at the level of compound flow graphs without inspecting the underlying activities. Therefore, it is useful to color compound flow graphs in much the same way as we did for activities. For this, we have to realize that the coloring of a compound flow graph cannot contain the same amount of information that is reflected by the coloring of all its children flow graphs. The coloring of a compound flow graph is always an abstraction of the coloring of its children flow graphs and their ports and channels. In cases where the state of a compound flow graph does not provide enough information, a designer will still have to inspect the internals of the compound flow graph.

A major problem in coloring compound flow graphs, is that not all children flow graphs in a compound flow graph are equally important in the view of a designer. It cannot be determined from a design flow configuration what designers experience as the dominant children flow graphs in a compound flow graph. For instance, should the compound flow graph shown in figure 5.25 be executable if a circuit can be generated, or if the stimuli can be edited, or perhaps only if both a circuit can be generated and a stimuli can be edited? This depends on the role these activities play in the context of the compound flow graph. The same holds for the question whether the compound flow graph should be considered executed if a stimuli description has been printed or only if a simulation has been performed and the simulation results have been viewed.

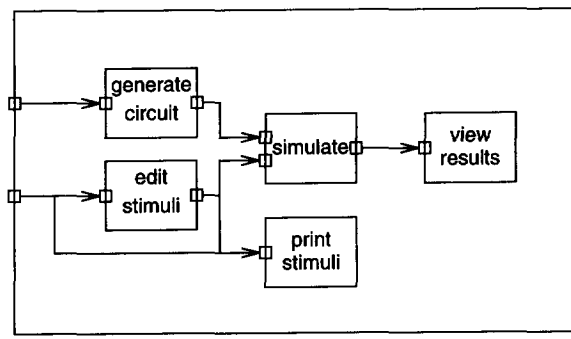


Figure 5.25. A compound flow graph.

It is our experience that the best strategy to handle this problem is to define a default rule for the coloring of compound flow graphs that takes the coloring of the children flow graphs and the way they are connected into consideration and, in addition, to allow the design flow configurator to fine-tune this coloring rule for each specific compound flow graph. This fine-tuning could be performed simply by specifying which children flow graphs need to be taken into consideration for the coloring of the compound flow graph. Of course, it is also possible to offer the design flow configurator an extensive language with logical expressions and primitives to refer to the coloring of the children flow graphs, so that he can 'program' the compound coloring for each individual flow graph. However, we think that the use of a proper default compound coloring rule in combination with the possibility to describe which children flow graphs should be taken into consideration offers enough flexibility to let the design flow configurator influence the coloring of compound flow graphs.

5.4 Design Scheduling

In many design environments a designer performs a restricted number of design actions or sequences of design actions multiple times on different (versions of) similar data. It would be useful if designers could define and execute these routine tasks, which we call *design schedules*, in a simple way. We capture this facility under the name *design scheduling*. The reason to describe design scheduling in this chapter is that since design schedules are to be defined, redefined and executed in the course of a design process, we like design scheduling to be supported by the design flow user interface.

In a way, a design schedule corresponds to the explanation an experienced designer would give an unexperienced designer on how to proceed. It specifies a preferred way of executing a number of activities in a predefined order. An illuminating way to look at the difference between a design flow and a design schedule is by comparing it with the difference between the *rules* and the *tactics* of a game. The rules of a game (design flow) specify what is *allowed*. The tactics to be practiced (design schedule) specify what is *preferred*. During a game, the rules remain unchanged, but the tactics could change, depending on the progress that is made and the actual status of the game.

An example of a design schedule that could be defined with respect to the design flow configuration in figure 5.26 is: "first expand a layout, then run the extractor on the expanded layout with capacitances taken into consideration (extract -c), then expand the derived circuit description and finally simulate the circuit with a stimuli description".

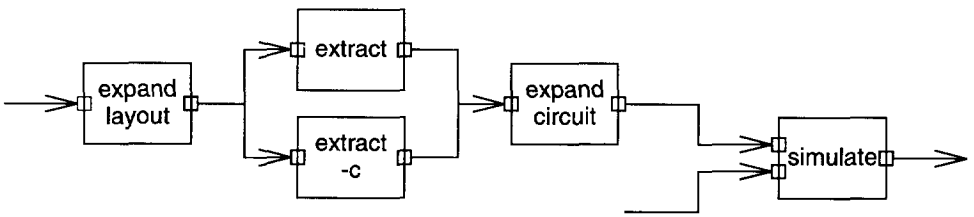


Figure 5.26. Design flow configuration for layout verification.

A straightforward way to support design schedules in a design flow system that allows design flows to be described hierarchically, is to define a design schedule as a compound flow graph. However, we feel that both mechanisms have a different use. A compound flow graph is part of the design flow configuration, which usually does not change during the design process. Design schedules are more dynamic in the sense that they can be (re)defined at any moment in the

design process. We feel that it is not desirable to use the design flow hierarchy for design scheduling. We need a mechanism that more closely corresponds to the specific characteristics of design schedules.

The main problem we face in this chapter is how to support design scheduling in a design flow user interface that uses colored flows to interact with the designers. On the one hand, the use of colored flows to specify the start conditions for design schedule execution and to visualize the execution of a design schedule is quite intuitive to the end-user. On the other hand, the fact that a colored flow shows only part of the design state may complicate the definition or execution of certain complex design schedules. Before we describe schedule definition and execution, in the next section, we first take a closer look at the information structure of design schedules. Design schedule definition and execution are addressed in sections 5.4.2 and 5.4.3 respectively.

5.4.1 Modeling Design Schedules

According to the informal example presented in the previous section, a design schedule is described by the activities that must be run in a certain order and with certain arguments. We name such an activity run that is to be performed during the execution of a design schedule a *scheduled activity run* and we call the arguments that are to be used upon the invocation of such an activity *scheduled arguments*.

Although the description of a design schedule via a number of scheduled activity runs may be sufficient to handle simple cases, there are situations where a more extensive mechanism is needed. For instance, consider the example in figure 5.27. Activity *c* has two input ports via which data is consumed that must be generated by activity *b*. In this design flow a designer may want to define a design schedule where activity *b* runs twice and activity *c* runs on the data entities that were generated by the two different runs of activity *b*. To handle this properly, the design schedule should, in addition to the scheduled activity runs, also specify which data must be accessed via which ports. We name the access that is to be performed during the execution of a design schedule *scheduled data access*.

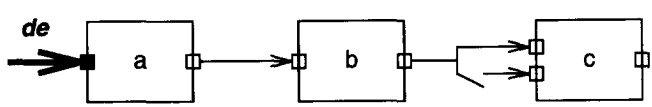


Figure 5.27. Design flow where a design schedule must describe scheduled data accesses.

Figure 5.28 shows the information model for design schedules. A *DesignSchedule* consists of a number of *ScheduledActivityRuns*, which define the *Activities* to be executed to achieve the desired design goal. For each *ScheduledActivityRun* there may be a number of *ScheduledArguments*, which define some *Instantiated-Arguments* for the activity invocation, and a number of *ScheduledDataAccesses*, which specify that a *ScheduledDataEntity* is to be accessed via a *Port* of this activity. Thus, we observe the static constraint:

ASSERT ScheduledDataAccess ITS ValidScheduledDataAccess (TRUE) =
  Port ITS Channel ITS FlowGraph =
    ScheduledActivityRun ITS Activity ITS FlowGraph

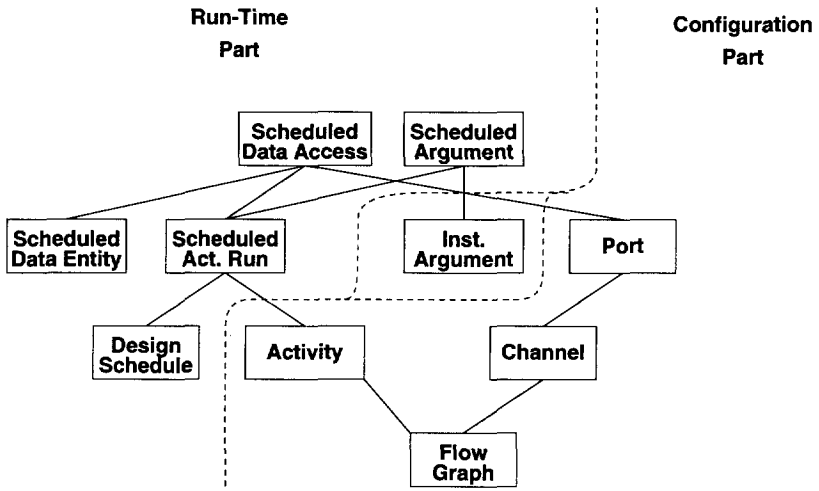


Figure 5.28. Information model for design schedules.

In the information models in chapter 3 and 4, we distinguished between object types that define information that changes during the design process (the *run-time* part) and object types that define information that does not change during the design process (the *configuration* part). Since design schedules may be (re)defined during the design process, they fall into the category of run-time information.

5.4.2 Defining Design Schedules in a Colored Flow

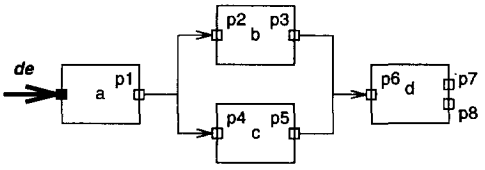
From the information model in the previous section we can see that the definition of a design schedule takes quite some effort. Designers need to indicate the activities to run, the arguments to use and the data to access. Without additional help this would seriously discourage the use of design schedules. However, a design flow user interface, knowing the structure of the design process from the design flow and knowing the history of the design process as administered in the run-time information, may assist designers with the definition of their design schedules in two ways. These are:

- *goal-oriented schedule definition*:
  In this case the design flow user interface derives a design schedule from the specification of a design *goal* in a colored flow. The goal may be expressed in terms of the activities that must be executed or the ports that must be colored. In this way a designer does not need to specify all individual activities to be run, he can simply specify what he is interested in and the user interface will do the rest.
- *history-oriented schedule definition*:
  In this case the design flow user interface derives a design schedule from part of the state of design shown in a colored flow. This allows designers to *replay* part of the design history on other data.

In *goal-oriented schedule definition* the design flow user interface uses the colored flow that is displayed as a specification of the context from which the design schedule should start. The design schedule that is defined may be executed on other data entities than the data entities that are displayed in the colored flow. The data entities in the colored flow are only used to specify where data is expected to be present when the design schedule executes. The design flow user interface calculates all intermediate activities that must be executed to fulfill the specified design goal. Due to the branches and merges in the design flow, this may result in multiple alternative design schedules.

An example of goal-oriented schedule definition is shown in figure 5.29. If the design flow user interface is in schedule definition mode and a designer selects the output port $p7$ of activity d, the design flow user interface will generate all design schedules that are expected to generate data on port $p7$, starting from the colored flow that is displayed. These design schedules, $ds1$ and $ds2$, are shown in the table of figure 5.29. This table has been obtained via the OTO-D query:

GET ScheduledDataAccess ITS ScheduledDataEntity, ScheduledActivityRun,
ScheduledActivityRun ITS Activity, Port,
ScheduledActivityRun ITS DesignSchedule



Scheduled DataEntity	Scheduled ActivityRun	Activity	Port	Design Schedule
sde1	sar1	a	p1	ds1
sde1	sar2	b	p2	ds1
sde2	sar2	b	p3	ds1
sde2	sar3	d	p6	ds1
sde3	sar3	d	p7	ds1
sde4	sar3	d	p8	ds1
sde1	sar1	a	p1	ds2
sde1	sar2	c	p4	ds2
sde2	sar2	c	p5	ds2
sde2	sar3	d	p6	ds2
sde3	sar3	d	p7	ds2
sde4	sar3	d	p8	ds2

Figure 5.29. Goal-oriented schedule definition.

To visualize a design schedule in a colored flow, we use the same techniques as we used for visualizing 'real' data entities and activity runs in a colored flow. We do not explain the flow coloring for schedule information in detail as we did for the coloring with run-time information, since the idea is very comparable. To visualize design schedules in a colored flow, we color ports and channels with scheduled data entities and activities with scheduled activity runs. For instance, figure 5.30 shows the representation of design schedule *ds* 2 in the colored flow of figure 5.29. Channels and activities colored with scheduled data entities and scheduled activity runs are drawn dashed and thick. Ports colored with scheduled data entities are drawn solid gray.

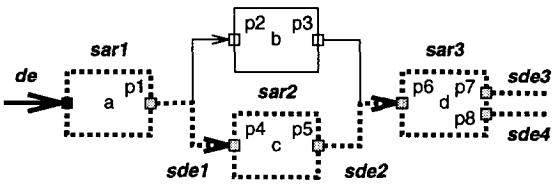
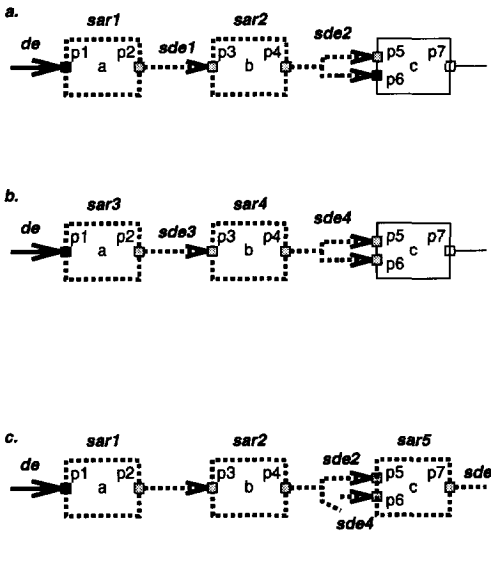


Figure 5.30. A colored flow with design schedule *ds* 2.

For defining more complicated design schedules, it must be possible to assign scheduled data entities and scheduled activity runs to a colored flow in the same way as the assignment of 'real' data entities and activity runs. For instance, in the example of figure 5.27 the designer must be able to assign two different scheduled data entities to the inputs of activity *c*. Figure 5.31 shows how this design schedule can be defined. The designer must choose activity *b* as a subgoal once (figure 5.31.a), clear the scheduled data entities *sde* 1 and *sde* 2 from the colored

flow, choose activity *b* as a subgoal again (figure 5.31.b), assign the scheduled data entities *sde* 2 and *sde* 4 to the inputs of activity *c*, and finally select the output port of activity *c* (figure 5.31.c).



Scheduled DataEntity	Scheduled ActivityRun	Activity	Port	Design Schedule
sde1	sar1	a	p2	ds1
sde1	sar2	b	p3	ds1
sde2	sar2	b	p4	ds1

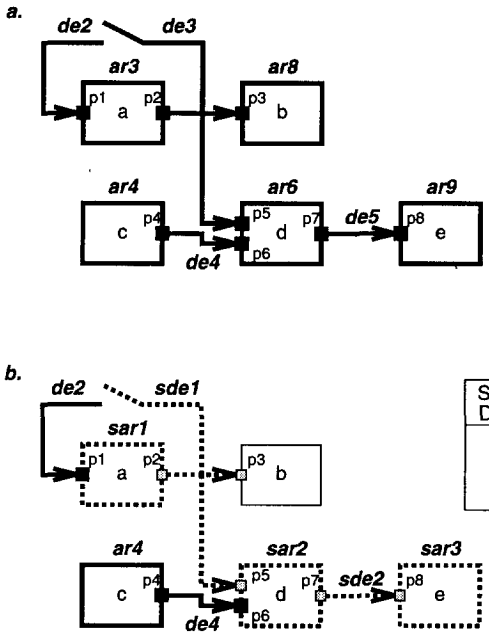
Scheduled DataEntity	Scheduled ActivityRun	Activity	Port	Design Schedule
sde1	sar1	a	p2	ds1
sde1	sar2	b	p3	ds1
sde2	sar2	b	p4	ds1
sde3	sar3	a	p2	ds1
sde3	sar4	b	p3	ds1
sde4	sar4	b	p4	ds1

Scheduled DataEntity	Scheduled ActivityRun	Activity	Port	Design Schedule
sde1	sar1	a	p2	ds1
sde1	sar2	b	p3	ds1
sde2	sar2	b	p4	ds1
sde3	sar3	a	p2	ds1
sde3	sar4	b	p3	ds1
sde4	sar4	b	p4	ds1
sde2	sar5	c	p5	ds1
sde4	sar5	c	p6	ds1
sde5	sar5	c	p7	ds1

Figure 5.31. Defining a complicated design schedule.

In *history-oriented schedule definition* the design flow user interface uses the run-time information shown in the colored flow as a specification of the design history to be replayed. Since the design flow system knows the history of the design process as administered in the run-time information, it can help designers defining such design schedules. Starting from a colored flow a designer may select a number of data entities and activity runs of which the history must be transformed into a design schedule. The design flow user interface inspects the recorded data accesses of the design data entities and the activity runs in which they were involved, and derives a design schedule.

Figure 5.32 shows the use of history-oriented schedule definition using the example of the beginning of this chapter. If the design flow user interface is in schedule definition mode and the designer selects the data entities *de* 2 and *de* 6 and activity runs *ar* 2 and *ar* 7 in figure 5.32.a, indicating that the design flow user interface should derive a design schedule that replays the functionality of this part of the design state, the design flow user interface will derive a design schedule *ds* 1, which is shown in figure 5.32.b.



DataEntity	ActivityRun	Activity	Port
de1	ar1	a	p2
de1	ar2	a	p1
de2	ar2	a	p2
de2	ar3	a	p1
de3	ar3	a	p2
de4	ar4	c	p4
de1	ar5	b	p3
de3	ar6	d	p5
de4	ar6	d	p6
de5	ar6	d	p7
de2	ar7	d	p5
de4	ar7	d	p6
de6	ar7	d	p7
de3	ar8	b	p3
de5	ar9	e	p8

Scheduled DataEntity	Scheduled ActivityRun	Activity	Port	Design Schedule
sde1	sar1	a	p2	ds1
sde1	sar2	d	p5	ds1
sde2	sar2	d	p7	ds1
sde2	sar3	e	p8	ds1

Figure 5.32. History-oriented schedule definition.

During design schedule definition certain assumptions are made about the data accesses that will be performed during an activity run. For instance, required output ports are supposed to generate data, while optional output ports do not necessarily produce data. The design flow user interface may use this information to generate design schedules that have a high chance to achieve the design goal selected. A logical implication of this choice is that the use of optional ports by the design flow configurator influences the schedule definition capabilities of the design flow user interface. In general, if the design flow configurator defines a design flow with many optional ports, and therefore gives the design flow system vague information about the behavior of the design activities, the design flow user interface will have more trouble in defining reliable design schedules, since it cannot be sure about the exact results of tool executions.

5.4.3 Executing Design Schedules in a Colored Flow

To visualize the execution of a design schedule in a colored flow, scheduled activity runs and data accesses must be mapped onto 'real' activity runs and data accesses and the corresponding run-time information should be shown on the appropriate flow primitives. This means that for all scheduled activity runs of a design schedule, the data entities that correspond to the scheduled data entities to be accessed in this scheduled activity run should be assigned to the colored flow, and the activity belonging to the scheduled activity run should be executed on these data entities.

As an example, figure 5.33 shows the execution of design schedule *ds* 1 as defined in figure 5.32 in a colored flow with data entities *de* 1 and *de* 4. In figure 5.33.a scheduled activity run *sar* 1 has been performed on data entity *de* 1, resulting in an activity run *ar* 10 that produced data entity *de* 7. Figure 5.33.b shows the design state after performing scheduled activity run *sar* 2.

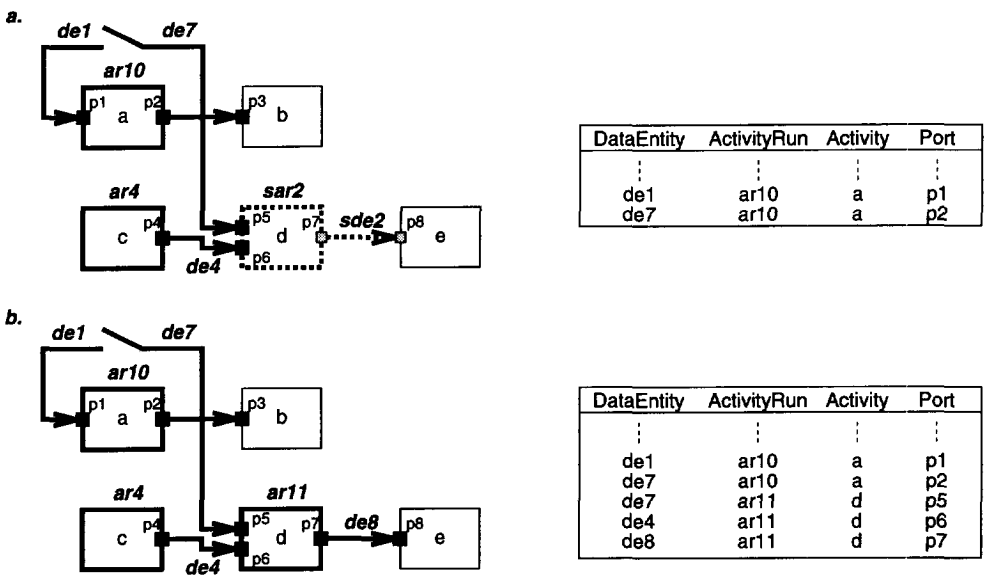


Figure 5.33. The execution of a design schedule in a colored flow.

As mentioned before, during design schedule definition certain assumptions must be made about the execution of an activity. For instance, a scheduled activity run may be supposed to produce a certain data entity. During design schedule execution, these assumptions may not be fulfilled. The actual operation of a CAD tool may depend on design decisions taken by designers while operating the tool, tool command line options, the environment from which the tool has been

invoked, the state and contents of certain files or the behavior of other CAD tools or designers in a distributed multi-tasking multi-user environment. It is this unpredictability that makes that the execution of a design schedule may terminate prematurely. Nevertheless, we think that the ability to define and execute design schedules in a user interface that is also used to browse through the design state and to invoke design tools, is a further step towards a smart design system.

5.5 Comparison with Other Approaches

In this section we compare the design flow user interface presented with a number of other approaches.

5.5.1 Monitor

In [Janni86] a user interface is presented that uses a mechanism comparable to the flow coloring mechanisms presented in this chapter. Central in this approach is the use of Petri nets to model a design process and to present the design state to the end-user. Tools, described by transitions in the Petri net, are 'colored' green, red or yellow, to show that they are executable, not executable or disabled.

An important difference between the monitor user interface and the user interface presented in this chapter, is that the monitor user interface does not make any selections to present meaningful parts of the design state to the end-user. It simply visualizes all tool runs and all data in one Petri net representation. This is possible since it invalidates all derived data if a tool that already has been executed is executed again. The design flow user interface presented in this chapter is conceptually different from the Monitor system in the fact that it can make meaningful selections from the design state stored in the underlying meta data database to present this information to the end-user in the form of a colored flow.

5.5.2 Roadmap

In the Roadmap system, described in [Hamer90], a flow-based user interface has been described that uses a representation based on a 'flowmap' to visualize the design state and to allow tools to be invoked on design data. A browsing technique is described that is comparable to the browsing operations described in this chapter. If the end-user selects a number of 'datasets' the user interface shows all related datasets.

A difference between the representations used in the Roadmap user interface and colored flows as presented in this chapter is that in the Roadmap representation runs are not shown explicitly, only the datasets involved in a run are shown. This

may be a problem for runs in which no data is produced. Another difference is that in the Roadmap approach the user has to select each individual dataset to be shown in the user interface from a set of possible datasets. In contrast, one of the essentials of the user interface presented in this chapter is that it automatically derives a colored flow from a single user-assigned data entity.

5.5.3 Hercules

In [Sutton93] the Hercules user interface has been described, which uses a *task graph* for design state presentation, tool invocation and for scheduling design operations. As stated before, a task graph is a combination of a design trace and a colored flow. A task graph may be expanded in any direction according to the relationships defined in the task schema.

With respect to design scheduling, the expansion of a task graph to schedule design operations may be compared with the definition of a design schedule in a colored flow as presented in this chapter. We have the feeling that the definition of a design schedule in a colored flow is somewhat simpler, because no expansion is needed to specify the design goal. On the other hand, the use of a colored flow for defining design schedules has the drawback that it may be necessary to specifically assign scheduled data entities to the colored flow since flow primitives cannot be replicated (as explained in section 5.4.2). The reason that we have chosen not to expand design flows for design scheduling, as in the Hercules approach, is that we think that the use of one stable description of the design process from which all user interaction takes place, has a higher priority than a slightly easier definition of design schedules.

5.5.4 The Texas Instruments Flagship Design System

A rather different view on user interaction for design flow systems has been described by Rumsey and Farquhar [Rumsey92]. They propose the use of a different 'user view' for different classes of design processes or design processes in a different state. This would be particularly important if the project crosses different engineering disciplines. They provide a different user view for situations where a designer executes many tools on a small set of data, situations where a designer executes a single tool on a large set of data and situations where a designer is concerned with the execution of a well defined process.

The view of Rumsey and Farquhar contradicts with the philosophy of the system presented in this chapter as well as with the other approaches in this comparison section. These approaches focus on the development of a single method of user interaction suitable for all types of applications. The first two user views of

Rumsey and Farquhar actually correspond with a purely data-oriented and a purely tool-oriented method of user interaction. In our opinion, these methods of user interaction are actually more limited than the flow-based method of user interaction. Therefore, we do not agree with the view of Rumsey and Farquhar. We think that flow-based user interaction is a powerful paradigm that can be used for all types of design applications in all states of the design process.

5.6 Conclusion

In this chapter we have presented a design flow user interface that exploits the information administered by the design flow system to assist designers during the design process. We proposed the use of a *colored flow* as a representation from which all user interaction can be performed. The main reason to prefer the use of a colored flow to the use of a *design trace* or a combination of a design trace and a colored flow is that a colored flow offers the designers a stable view on the design process that only changes when design tools are added or removed, and that this representation technique allows the visualization of what *has been performed* (the design state) as well as what *can be performed* (the applicable design actions).

A colored flow represents part of the state of the design process in a design flow, which is actually a description of the same design process. We claim that the flow coloring techniques presented in this chapter can be used to create a user interface that assists designers in performing all their basic design actions.

The first design action mentioned, *browsing through the design state*, is supported by the simultaneous presentation of data entities and activity runs in a colored flow. The *browsing operations* for colored flows help designers to retrieve information about derivation relationships between data entities and activity runs. The *flow coloring operations* can be used to adjust a colored flow to the designer's interest. In addition, the automatic derivation of a *maximal colored flow* that visualizes the derivation context of a user-assigned data entity minimizes the efforts needed from a designer to let the user interface focus on interesting parts of the design. The executability state assigned to each activity and compound flow graph and the coloring of each activity and compound flow graph help designers to select an activity of a tool. They can easily see whether an activity is executable and whether it has already been executed with respect to the data entities that color its input and output ports.

Although not specifically addressed in this chapter, the *invocation of a design tool* can be supported by automatically building a default command line from the selection of an activity in a colored flow. The command line to be put together

should refer to the data entities that color the input ports of the activity. Since the design flow system knows about the structure and meaning of the different arguments of the activities (as specified in chapter 3), the design flow user interface may inform the designers of the details of the tool invocation command line. For instance, designers may choose to invoke the tool with the default command line, or to interactively inspect and change the options and parameters via a tool invocation form. For these options and parameters, the user interface may supply meaningful default values based on previous tool invocations.

The *operate design tool* phase can be supported by updating the colored flow upon the completion of each activity run. In this way the execution of a tool can be inspected while the tool is still running. We address this issue in more detail in the next chapter, where we present the design flow user interface of the Nelsis CAD framework (see section 6.4.1).

The *organization of data* is supported by allowing the invocation of methods for data organization on data displayed in a colored flow. Since a colored flow shows a graphical abstraction of data entities, it can be used to let designers perform operations for data organization, like 'change status', 'remove data', 'rename data', etc. on these data entities. By configuring different methods for data organization for different parts of the design flow, the user interface can be tuned for a specific design application.

The fact that all the basic design actions can be performed from one and the same flow-based representation, makes it easy for designers to switch from one design action to another.

To facilitate the execution of a sequence of design actions in a simple way we presented the notion of *design scheduling*. We described how design schedules can be defined and executed in a colored flow. Design schedules are particularly useful to execute routine tasks or to replay part of the design history.

6

Implementation in Nelsis

6.1 Introduction

In the previous chapters we made few assumptions about the way design tools are integrated into a common operating environment. The most important assumption we made is that design tools communicate with a CAD framework through a procedural interface that offers at least functions to read and write design data from and to a data storage component. A CAD framework that satisfies this requirement is the Nelsis CAD framework, developed at Delft University of Technology. In this chapter we describe the implementation of the general design flow concepts in the Nelsis CAD framework.

To optimally exploit the general design flow concepts presented in the previous chapters in a Nelsis-based design environment, we have to tune them to the specifics of the Nelsis CAD framework. For this, we follow the structure of this thesis. First we refine the general tool model presented in chapter 3, then we extend the design flow model presented in chapter 4 and finally we apply the concepts for user interaction presented in chapter 5 to the general design system user interface of Nelsis. In addition, we describe how changing design environments can be handled properly, we address the architectural aspects of the integration of design flow management into Nelsis and we briefly address the cooperation of the design flow system with two other services of the nelsis CAD framework: library management and access control. Finally, at the end of this chapter we describe the use of the Nelsis design flow system in a number of different application domains.

6.2 The Nelsis Tool Model

The general tool model presented in chapter 3 addresses two types of tool characteristics: data access characteristics and tool control characteristics. In this section we examine for both types of tool characteristics whether it is useful to extend the general tool model with Nelsis-specific constructs and if so, how. First we briefly describe the Nelsis data organization. Then, we give an overview of the Nelsis methods for data access. After that, we refine the data access description part of the general tool model for use in the Nelsis CAD framework. At the end of this section we spend a few words on the use of TES for describing how to control design tools in Nelsis.

6.2.1 Data Organization in Nelsis

In Nelsis, design data is distributed across *design projects*. The organization of design data in design projects yields a logical distribution of design data and design activities among different databases and design teams. A design project offers designers a local working environment for a number of design activities with a common goal. Multiple designers may work concurrently in one and the same design project. Within a design project, design data is organized into *design objects*. Each design object is managed by the CAD framework as a single logical unit of design data. A design object has a number of attributes, such as a *name*, a *version number* and a *viewtype*, which describes its representation. Design objects may be related through different types of inter-design object relationships, such as version relationships, which describe the versioning history of a design, and hierarchy relationships, which describe the hierarchical decomposition of a design. Conceptually, Nelsis does not make any assumptions about the data organization within design objects. The data within a design object may have any structure and may be stored in any type of data storage component, from an object oriented database to ordinary files. However, the default storage component that is delivered with the system stores the design data within a design object into a number of *streams*. For a more extensive discussion of the data organization in Nelsis, see [Wolf94a].

6.2.2 Data Access in Nelsis

Tools integrated into the Nelsis CAD framework access their design data via a procedural interface, called the *data management interface (dmi)*. The dmi-functions are called either by the design tools directly (integrated tools) or by their wrappers (encapsulated tools). To adhere to the organization of design data into design projects, design objects and streams, the dmi has a layered structure. To

access a particular stream in a particular design object in a particular design project, a tool must acquire access permissions for the design project, the design object and the stream respectively. Although not all the layers of the dmi have the properties of a transaction in the classical sense, the layered structure of the dmi has been called a *layered transaction schema* [Meijs87, Widya88].

Figure 6.1 shows a graphical representation of the layered transaction schema of the Nelsis dmi. This graphical notation is a variation of the graphical notation defined by Jackson [Jackson83]. Each rectangular box with rounded corners represents an action. A line connecting the bottom of an action to the top of another action specifies that the lower (child) action is performed within the upper (parent) action. The ordering of children actions from left to right indicates the chronological order of execution within its parent action. An action with a star in the upper right corner may be performed multiple times within its parent action.

Figure 6.1 shows that a *tool run* is bracketed by an *initialize* and a *terminate* action. The initialize action is used to inform the CAD framework of the beginning of a tool run. The tool identifies itself via its name and command line options. Using the terminate action a design tool informs the CAD framework of the end of a tool run, so that the CAD framework can perform the necessary cleanup operations.

Within a *tool run* a tool may perform any number of *project transactions*. A project transaction comprises a limited amount of work to be performed within the context of one specific design project. Using an *open project* action with a specific *mode* a design tool specifies which type of access it is going to perform on the particular design project. The *close project* action notifies the CAD framework of the end of the operations to be performed in the project.

Within the scope of a project transaction, a tool may perform any number of *design transactions* on design objects residing in this design project. A design transaction is initiated by a *check out* action with a specific *check out mode*, which specifies the type of access that is to be performed on the design object. If a tool checks out a specific design object, the design object is locked and the tool may work on the design data stored in the design object for a possibly longer period of time. Upon the termination of a design transaction, using a *check in* action, a tool may specify whether the changes to the design object are made persistent (commit), or whether the state of the design object is rewinded to the state corresponding to the situation before the transaction (abort). This implies that design transactions are atomic in the sense that they either succeed completely or fail completely.

Within a design transaction any number of *stream transactions* may be performed to create, read, write or destroy streams. A stream transaction is bracketed by an *open* and a *close stream* action.

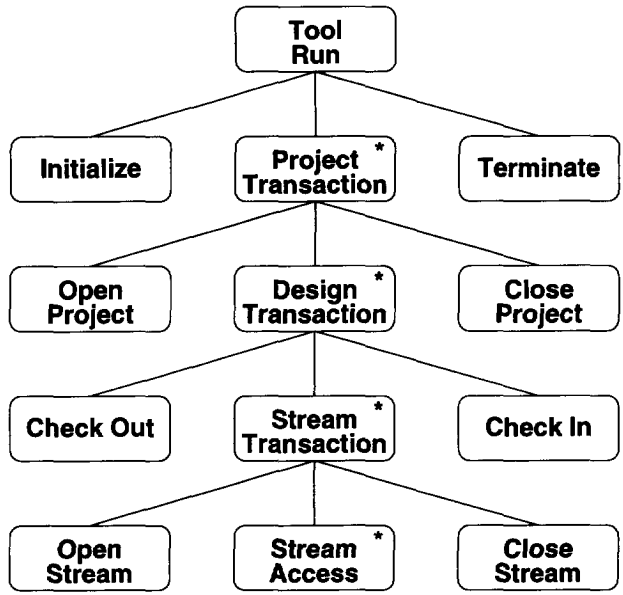


Figure 6.1. Transaction schema of the Nelsis dmi.

The Nelsis transaction schema describes data access in the Nelsis CAD framework from the perspective of the design tools: it specifies a *tool view* on data access. The Nelsis tool model on the other hand should describe data access from the perspective of the design flow system: it should reflect a *flow view* on data access. Due to these different points of view, it would be wrong to simply transform the nelsis transaction schema into a Nelsis tool model. The Nelsis transaction schema has been tailored to an efficient and easy to program data access of the design tools while the Nelsis tool model should be optimized to describe those data access characteristics of design tools that are important for design flow management.

In the sequel of this section we combine the general tool description concepts presented in chapter 3 with the Nelsis-specific transaction schema as described above, to arrive at a tool model for describing data access characteristics of design tools integrated in Nelsis. When comparing the general tool model (figure 3.5) with the Nelsis transaction schema (figure 6.1), we observe three important differences. First of all, the general tool model does not address project transactions, second, the Nelsis transaction schema does not address activity runs

and third, it is unclear how Nelsis design transactions and Nelsis stream transactions relate to the data accesses as used in the general tool model. Below, we describe each of these aspects in more detail.

6.2.2.1 Projects

The general tool model as presented in chapter 3 does not have any constructs to describe project access characteristics of design tools. Although it is very well possible to extend the tool model with such constructs, it is questionable whether this is desirable from a design flow point of view. This depends on the actual use of design projects in Nelsis.

The purpose of the distribution of data and design activities among a number of logically independent design projects in Nelsis is to create a manageable and orderly working environment for both designers as well as the CAD framework. Designers typically work in the scope of a single design project for a longer period of time and the tools they invoke typically operate on design objects belonging to this particular design project. In situations where a tool does operate in different design projects within a single tool run, it typically accesses data residing in a so-called *library project*[1]. The observation that we like to make here is that tools invoked by a certain designer operate most of their time in the project this designer is working in and that the operation of the tools in other projects typically is limited to inspecting some data residing in library projects.

One of the basic concepts in our design flow system is that we express the behavior of a tool in terms of units of tool operation meaningful to the designers. From the above, we think we may conclude that such a meaningful unit of tool operation is always performed within the context of one project. In other words: although a tool may operate in multiple projects, the activities it performs do not cross the project borders. The implication of this observation is that, for the description of the activities of a tool, we do not have to describe the project access characteristics of the tool.

In addition, since design projects provide designers with a local context in which most of their design activities take place, they are typically not interested in the behavior of a design tool in other projects than the one currently working in. Therefore, we choose to describe the data access characteristics of the design tools

1. We explain the use of library projects in more detail in section 6.7.1.

and to administer the run-time information *per project*. In fact, this choice has also been inspired by the fact that the Nelsis CAD framework administers its data management information per project.

Summarizing, we do not extend the tool model as shown in figure 3.5 with a construct to describe the project access characteristics of design tools and we let the object type *Tool Run* in this information model actually correspond to a *Project Transaction* in the Nelsis transaction schema.

6.2.2.2 Activities

The second difference between the transaction schema of the procedural interface of Nelsis (figure 6.1) and the general tool model as presented in chapter 3, is that the object type *Activity Run* in the general tool model has no counterpart in the transaction schema. This is not only a question of using different names. An activity run as introduced in chapter 3 is not the same as a design transaction since multiple design objects may be accessed within one and the same activity run, and it is not the same as a tool run since we adopted the policy that multiple activity runs may be performed within one and the same tool run. The absence of the activity concept in the Nelsis transaction schema can be explained from the observation that it has been designed primarily for an efficient and easy data access (from the 'tool viewpoint') and not for reasons of design flow management (from the 'flow viewpoint'). After all, the main interest of the tools is to access their design data and not to inform the design flow system about the activities they perform. However, to perform fine-grain design flow management the design flow system must somehow find out which activity runs are performed during a tool run.

One way to solve this problem would be to extend the Nelsis procedural interface with a *start activity* and a *stop activity* action. This enables tools to explicitly notify the design flow system of the beginning and the termination of an activity run. However, the need to add these flow-specific calls to the source code of the design tools or their wrappers may be a hindrance to actually use the design flow system. It is our experience that tool developers do not like to be bothered with the activity runs of a tool as being administered in the design flow system of the CAD framework. Their primary interest is to let their design tools work. For this, they focus on the domain specific algorithms to be implemented and the communication with the underlying CAD framework is kept as small as possible and primarily focuses on how to get the data out of the data storage component and how to write it back.

Another argument against the use of a start and stop activity action is that when a tool starts to access data, it is not always known which of its activities it is going to perform. For instance, this may depend on user interaction. Since in the above solution tools are forced to notify the design flow system of the start of an activity run before they access any data, this would seriously hamper the freedom of programming of the tool programmers.

Another way to compensate for the absence of the activity transaction level in the procedural interface of Nelsis, is to *recognize activity runs* from the data accesses performed during a tool run. Since the activities of a tool differ in their data access patterns (observation 3.1), the design flow system should be able to recognize the individual activity runs by carefully analyzing the operation of the design tools. This makes the activity level a *virtual* level in the transaction schema: the activity level does exist from a design flow point of view but tool developers do not need to encode start and stop activity requests into their source code. Figure 6.2 shows the transaction schema with the virtual (dotted) activity run level.

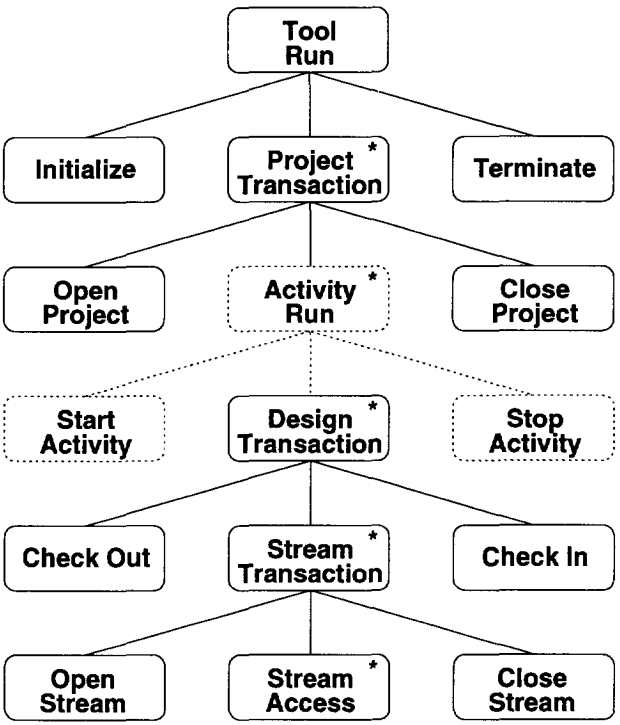


Figure 6.2. Virtual activity run level in the transaction schema.

To recognize activity runs from data accesses the design flow system must compare the data accesses of the activities configured for that tool with the data accesses actually performed. Since it is possible that a data access does not immediately identify one specific activity of a tool, the design flow system should register which activities are still a candidate to be recognized and which are not, taking into account the data accesses already performed. For this, it maintains a list of *candidate activities*, where a candidate activity is specified by the activity that possibly executes and the design objects[2] that already have been accessed. However, the administration of one list of candidate activities is not enough to correctly recognize all activity runs. Since a design tool may perform any of its activities any number of times during a tool execution (even concurrently), the design flow system should maintain multiple sets of candidate activities, called *candidate sets*, per tool run. At any moment the number of candidate sets for a tool corresponds to the number of activities of this tool running concurrently.

Having a number of candidate sets with possibly multiple candidate activities, the design flow system assigns each design transaction performed during a tool run to one particular candidate set. The characteristics of this design transaction, such as the access mode that is used, the viewtype of the design object that is accessed, are compared to the characteristics of the ports of the candidate activities in this candidate set. For each candidate activity there are two possibilities: if there is no port to which the design transaction can be assigned, the candidate activity is removed from the candidate set; if the design transaction can be assigned to n different ports, the candidate activity is duplicated $n-1$ times and the design transaction is assigned to each of the n duplicates to a different port. Upon tool termination each candidate set must contain one and only one candidate activity or multiple 'compatible' duplicates (i.e. duplicate activities which have the same design objects assigned to their ports, in different ordenings). If multiple candidate activities remain or if the candidate set is empty, the system failed to identify a particular activity and the activity definition of the tool or the tool itself should be adapted.

2. For simplicity, we restrict the explanation of the activity recognition process to the access to design objects only. However, in the actual implementation the stream accesses performed within a design transaction are also used for the recognition of activity runs.

Figure 6.3 gives a simplified example of the activity recognition process. It shows the transformation of a candidate set upon a check out request for design object *do* 3, issued by the design tool for which the candidate set is maintained. The initial candidate set contains three candidate activities, which each have the design objects *do* 1 and *do* 2 assigned to their ports in a different way. Upon the check out request for design object *do* 3, the design flow system removes candidate activity 1 from the candidate set, since it has no port to which design object *do* 3 can be assigned. Candidate activity 2 contains exactly one port to which design object *do* 3 can be assigned and therefore the design flow system simply copies it to the new candidate set and assigns design object *do* 3 to the appropriate port. Since design object *do* 3 can be assigned in two different ways to candidate activity 3, this candidate activity is duplicated once. This results in candidate activities 3 and 4 in the new candidate set.

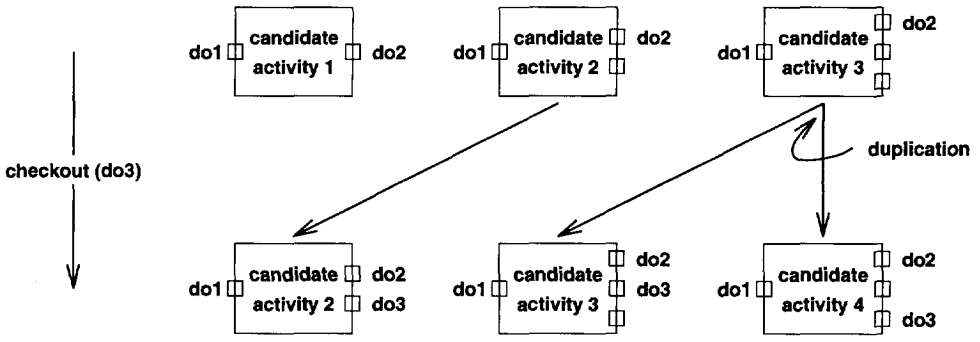


Figure 6.3. Transformation of a candidate set upon a check out.

A side-effect of the recognition of activity runs from data accesses is that the actual identification of a running activity may not happen until an identifying data access is performed. Especially in cases where multiple almost similar candidate activities exist, the identification of an activity run may take some time. As long as there is any chance that the tool is going to perform an activity that it is allowed to perform according to the constraints specified in the design flow configuration, the design flow system should allow the further execution of the design tool. This means that if, later on, none of the candidate activities appears to be identified, it may be that a number of design transactions have already been performed, which actually were not allowed from a design flow point of view. This problem can simply be solved by using a rewind mechanism at the level of activity runs, to undo those design transactions that appeared to belong to an activity that actually was not allowed to be executed or to postpone the actual completion of a design transaction until the activity it belongs to has been identified and granted.

Another problem with the activity recognition process may be that the duplication of candidate activities explodes exponentially with respect to the number of similar ports of the activities. However, this is not really a problem as long as the number of ports of the activities is limited or if the characteristics of the different ports differ considerably.

All things considered, we feel that preference should be given to the recognition of activity runs from data accesses over extending the Nelsis transaction schema with a start and a stop activity action. The latter solution would come down to adapting the 'tool view' to the design flow system, while the first solution preserves the well-defined and commonly agreed 'tool view' and solves the design flow management issue in the design flow system itself.

6.2.2.3 Design Objects, Streams and DataTypes

The third difference between the Nelsis transaction schema (figure 6.1) and the general tool model of chapter 3, is the existence of two different transaction levels for accessing data in Nelsis as opposed to one data access level in the general tool model. To tune the tool model for use in the Nelsis environment, we have to decide which of the different transaction levels to support in the Nelsis tool model. In other words, we must decide at which level to couple data to tools. For this, our objective is to model the data access characteristics of design tools in so much detail that the tool descriptions generated according to this model are detailed enough to enrich the design flow system with profound knowledge of the design tools, and restricted enough to enable an easy generation of tool descriptions.

The organization of design data into design objects is an important principle in Nelsis. Design tools operate on design objects, different versions of design data are stored in different design objects and hierarchical designs are described in terms of design objects and their hierarchical relationships. Therefore, it is straightforward to let the Nelsis tool model describe data access characteristics at the design object level. The question is however whether it is sufficient to describe tool characteristics at the design object level only or whether it is useful to describe the behavior of design tools in Nelsis at the level of streams as well. An advantage of such a detailed description is that it provides the design flow management system with very detailed information about the data access characteristics of the design tools, which may be important for a precise prediction of their executability.

Although in some cases the existence of individual streams within design objects is crucial information for determining the state of the design process, in other cases the existence of one of a set of streams that are generated and accessed together is not important at all. In the latter case, it is not useful nor desirable to configure the data access characteristics of design tools for each of the individual streams. In this situation it is sufficient to describe the tool characteristics in terms of sets of streams. It appears desirable to provide a mechanism that enables a detailed description of tool characteristics where necessary and that allows a lenient tool description in other cases. We take the position that it should be left to the tool programmer or design flow configurator to decide about the level of detail to be used for describing the data access characteristics of a tool.

An attractive solution to achieve such a flexible tool model is to think of streams as organized into abstract data of a certain *datatype* and to describe data access characteristics of design tools at the datatype level instead of at the streamtype level. By configuring the mapping of streamtypes to datatypes, the design flow configurator can choose the degree of detail for describing the data access characteristics of a set of design tools. The design flow system uses this mapping to recognize the access to data of a specific datatype from the observed accesses to streams. This solution leads to another virtual transaction level in the transaction schema of Nelsis: the data transaction level. Figure 6.4 shows the Nelsis transaction schema with the virtual (dotted) datatype level.

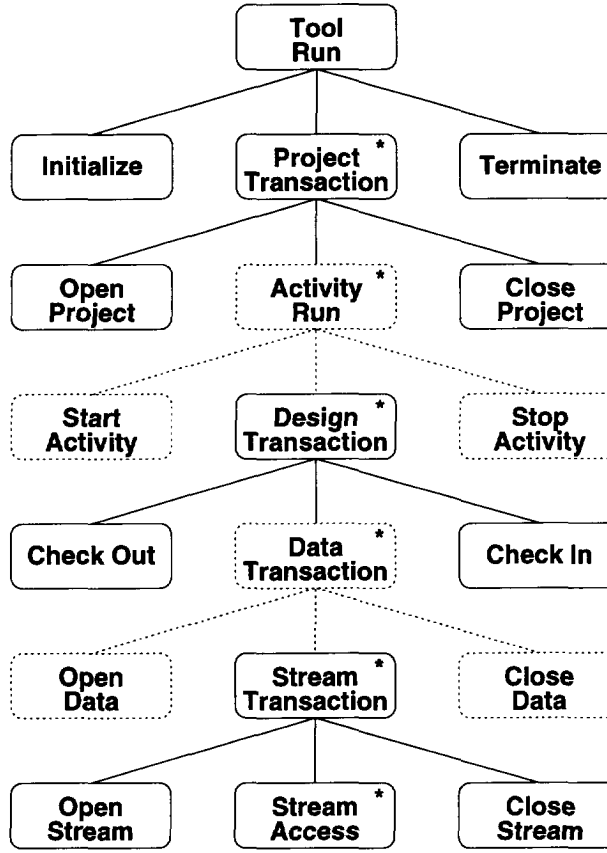


Figure 6.4. The Nelsis transaction schema with the virtual datatype level.

To describe data access characteristics of design tools at the datatype level as well as at the design object level, we have to split the port construct in the general tool model into two different types of ports: *design object ports* and *data ports*. Where a design object port describes a design transaction at the design object level, a data port describes the access to data of a specific datatype. Since the streams that make up data of a datatype are always accessed within a design transaction on a specific design object, each data port belongs to one specific design object port. Characteristics such as check out mode, stream access modes, optionality of access etc. are summarized into the *type* of each type of port. Since the access described by a data port must be in agreement with the access described by its design object port, there are certain constraints on the types of data ports belonging to a design object port. For instance, it does not make sense to try to read data of a datatype within the scope of the creation of a design object or to

create data of a datatype within the scope of a read access on a design object. These constraints can simply be checked at design flow configuration time. We do not explicitly administer the existence of data of a datatype within a design object, since this information can be derived from the data transactions that have been performed within the design transactions on that specific design object.

Figure 6.5 shows the resulting tool model. A *DOport* (*DesignObjectPort*) describes data access at the design object level and may have multiple *DataPorts* to describe data access at the data level. A *DOPort* has a *ViewType* to describe the type of design object that will be accessed via this port and a *DOPortType* to describe how the design object will be accessed. A *DataPort* has a *DataType* to describe the type of data that will be accessed via this port and a *DataPortType* to specify how the data will be accessed. A *DataTransaction* administers the access of design data of a specific *DataType* via a specific *DataPort*.

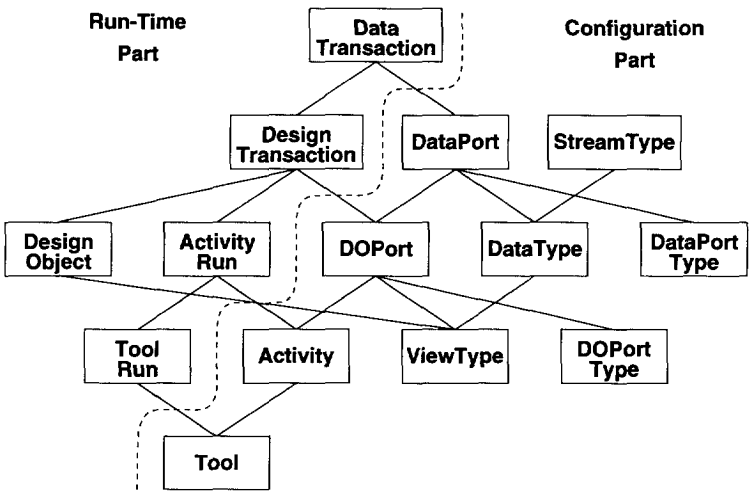


Figure 6.5. Data ports and design object ports describe data access at the data level and the design object level respectively.

The information model of figure 6.5 satisfies our original goal to let the design flow configurator choose his favorite level of detail for describing data accesses. Given a specific viewtype to streamtype mapping that is decided upon by a set of related design tools, the design flow configurator may position the datatype somewhere in between. This makes it possible to vary the level of detail in a data access description from the coarse-grain level of design objects (each viewtype contains at most one datatype) to the fine-grain level of individual streams within design objects (each datatype contains at most one streamtype).

6.2.3 Controlling Tools in Nelsis

Besides information about the data access characteristics of design tools, a tool description also contains information about how to invoke and operate a design tool. This is captured under the name tool control information. Tool control information for Nelsis design tools can be described quite well with the TES standard mentioned before (section 3.3). However, since TES is defined independently of a specific CAD framework such TES descriptions describe the tool control information of the design tools in a framework-independent way. Since it is a common situation that a command line, used to invoke a design tool integrated in Nelsis, contains certain Nelsis-specific arguments, such as a reference to a viewtype, a design project, a design object or a stream, we have to find a way to let framework-neutral TES descriptions refer to Nelsis-specific constructs. To support designers with invoking their design tools it is important that the tool descriptions explain the specific meaning of all command line arguments, including the Nelsis-specific arguments.

Fortunately, TES has a so-called *property* construct to add any CAD framework-specific information to the description of a command line of a tool. We use this construct to add Nelsis-specific information to the tool control part of a tool description. For instance, the property *nelsisproject* specifies that a certain command line argument identifies a nelsis project, the property *nelsisdesignobject* specifies that a command line argument identifies a nelsis design object and the property *nelsisviewtype* specifies that a command line argument identifies a nelsis viewtype.

A disadvantage of using the property construct to extend tool descriptions with Nelsis-specific information, is that it may hamper the interchange of TES descriptions among different design environments. Since the Nelsis-specific properties will be ignored in a non-Nelsis environment, the expressiveness of the Nelsis-TES descriptions will be weakened in other environments. It would be better if the general purpose TES language would have constructs that describe such information in general terms. This would reduce the need to define framework-specific properties at various sites in the world, which may frustrate the overall goal of CFI, the world-wide plug and play of design tools in different design environments. However, as long as TES does not contain enough expressiveness to describe the tool control characteristics of design tools integrated in Nelsis, the use of the property construct is a satisfying alternative.

6.3 The Nelsis Design Flow Model

The design flow model presented in chapter 4 addresses the modeling of data and temporal dependencies in a design flow, the hierarchical decomposition of a design flow and the configuration of additional information on the design process using port relationships. For the implementation of the design flow system into Nelsis we examine which of these constructs need to be refined and how.

6.3.1 Modeling Dependencies in Nelsis

In the general design flow model data and temporal dependencies are described by channels connecting a number of input ports to a number of output ports. Since the Nelsis tool model contains two types of ports, design object ports and data ports, we have to decide whether to model dependencies at the level of design objects or at the datatype level or both. Since dependencies at the datatype level yield a more detailed description of the design process than dependencies at the level of design objects and since dependencies at the level of datatypes make the use of dependencies at the level of design objects superfluous, we choose to link datatype ports to channels. This results in an information model as shown in figure 6.6. For simplicity, we merged the *Virtuality* attribute of a data port (see figure 4.6), which is necessary to describe temporal dependencies, into the *DataPortType* of the data port.

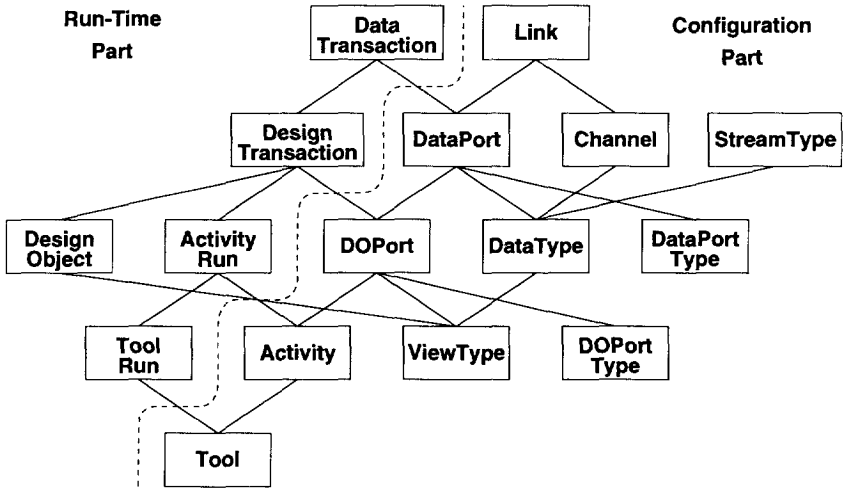


Figure 6.6. Dependencies are modeled at the datatype level.

Figure 6.7 shows an example of the use of dependencies at the datatype level. The two types of ports are visualized as nested rectangles. Data ports are drawn as

small rectangles contained in larger rectangles, which visualize the design object ports. The editor activity *a* and the activity *c* each produce design objects of a different viewtype. Activity *b* adds data of a certain datatype to design objects produced by activity *a*. Activity *d* reads the added data of a design object produced by activity *a* and a design object created by activity *c* and creates another design object with data of two different datatypes.

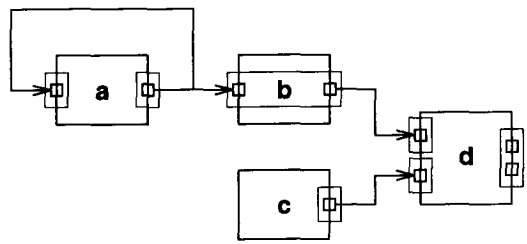


Figure 6.7. Design flow with design object ports and data ports.

6.3.2 Design Flow Hierarchy in Nelsis

For describing the hierarchical decomposition of a Nelsis design flow, we can use exactly the same method as described in section 4.3. This leads to an information model as shown in figure 6.8. For simplicity, we left out the administration of the runs of activity instances.

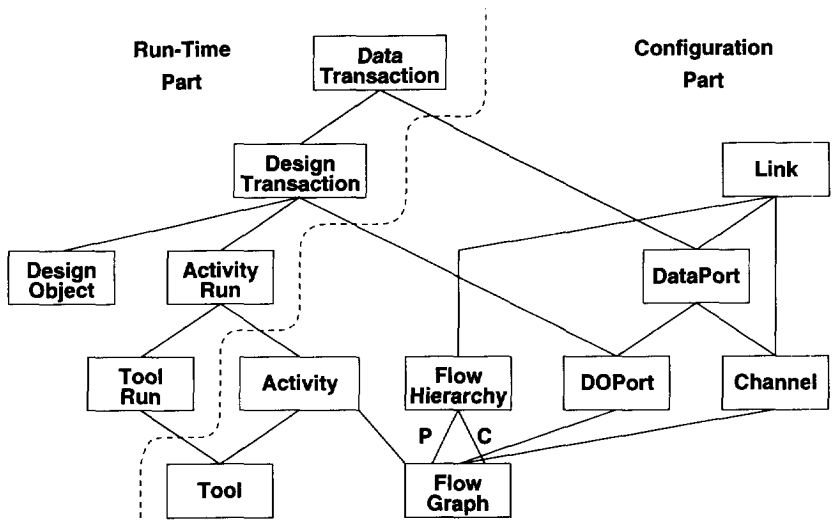


Figure 6.8. Modeling hierarchical design flows in the Nelsis design flow model.

6.3.3 Port Relationships in Nelsis

In chapter 4 we described port relationships as a way to configure constraints on the design process based on the interrelationships between design data entities. In Nelsis, interrelationships between design data entities are administered at the design object level. Design objects can be related through version relationships, hierarchical relationships and equivalence relationships. Version relationships define the version derivation of a design, hierarchical relationships define the hierarchical decomposition of a design and equivalence relationships can be used by design tools to administer a tool-specific relationship between two design objects. To refer to these inter-design object relationships from within a Nelsis design flow, the Nelsis design flow model can be extended with a port relationship construct at the level of design object ports. This is shown in figure 6.9.

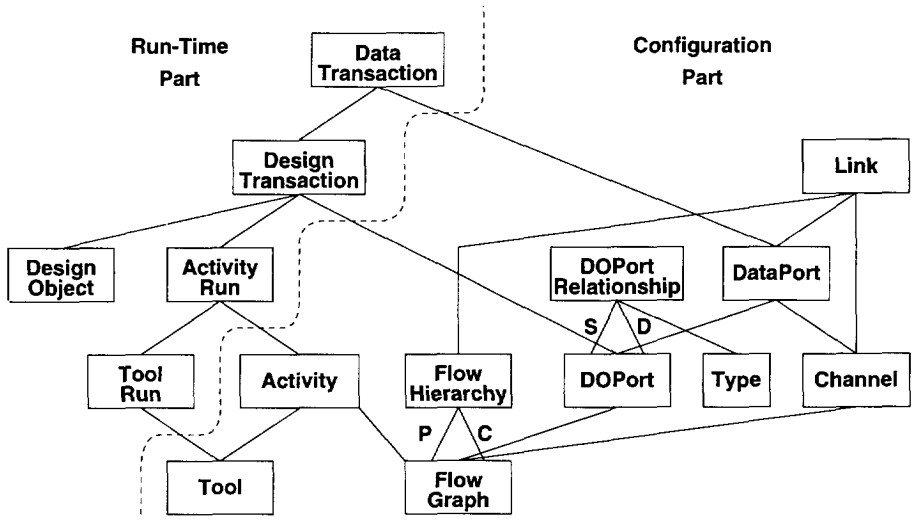


Figure 6.9. Port relationships are modeled at the design object level.

The design object port relationship construct is generic in the sense that it can be used to define constraints on the design process that relate to different types of inter-design object relationships as well as to define the automatic creation of specific inter-design object relationships. The *Type* of a design object port relationship determines its exact meaning.

An example of the use of a design object port relationship is to specify a requirement that the design objects accessed via source and destination design object ports must be hierarchically related (either directly or recursively). Another example is that a design tool may only execute on equivalent design objects.

Another use may be to configure the automatic creation of an equivalence relationship between two design objects accessed via the source and the destination port. Yet another application is to use design object port relationships to direct the derivation of a maximal colored flow, by indicating that certain ports may be colored with design objects that have a specific relationship only. A more detailed explanation of the use of design object port relationships in the Nelsis CAD framework (called object relationship constraints) can be found in [Wolf94b].

6.4 The Nelsis Design Flow User Interface

In this section we examine how the principles for flow-based user interaction as presented in chapter 5 can be applied to the user interface of the Nelsis CAD framework. In particular, we describe the implications of the use of the Nelsis-specific tool model and design flow model for deriving and representing a colored flow and for the definition, execution and representation of design schedules.

6.4.1 Flow Coloring in Nelsis

In chapter 5 we described how design flow primitives can be colored with run-time data to show the design state in the context of a design flow. In the Nelsis design flow model, activities can be colored with activity runs just as in the general design flow user interface but the coloring of ports and channels needs to be refined. It must be decided how to color the different types of ports and how to represent them to the end-user. To decide on the different methods of coloring a Nelsis design flow with run time data, we take a global look at how users interact with the Nelsis CAD framework.

The user interface of the Nelsis CAD framework, the *Design System User Interface (DSUI)*, presents information about the design process in terms of design objects and their relationships. For instance, its version browser displays the versioning of a design as a graph with nodes indicating design objects and edges indicating version relationships. Similarly, its hierarchy browser represents the hierarchical decomposition of a design by a graph of nodes indicating design objects and edges indicating hierarchical relationships. Upon the invocation of a design tool a designer specifies the data on which the tool is to be executed via the design object(s) in which this data resides. Summarizing, user interaction in Nelsis is dominated by the design object concept.

Although the Nelsis design flow model describes the design process in more detail than at the level of design objects, we do want the Nelsis design flow user

interface to ally to the general principles for user interaction as employed in the Nelsis CAD framework. Not only does this contribute to the unobtrusiveness of the design flow system, the consequent use of such a simple but powerful principle does also reduce the learning time of the system and therefore contributes to a global acceptance. Therefore, our general strategy is to let designers interact with the design flow user interface at the level of design objects while visualizing the design state at the detailed level of data within design objects. Since each data port belongs to exactly one design object port and the coloring of a data port implies the coloring of its design object port with the design object to which the data belongs, we choose to omit design object ports from the graphical representation and to color channels and data ports with design objects.

Figure 6.10 shows this method of coloring flow primitives for the example design flow of figure 6.7. The design object ports are simply left out from the picture and the data on data ports and channels is identified by the design object to which it belongs. Since design object ports play a minor role in the rest of this section, from now on, we simply refer to data ports as 'ports'. Although the output ports of activity *a* and activity *b* are colored with the same design object *do*2, their coloring refers to different data within this design object. The same holds for the two different output data ports of activity *d* that show data of different datatypes within one and the same design object *do*4.

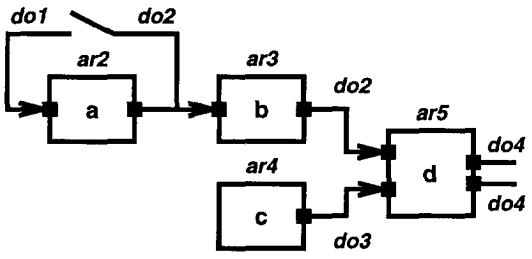


Figure 6.10. A colored flow as presented in the Nelsis design flow user interface.

As described in chapter 5 designers interact with the design flow system via a number of browsing and flow coloring operations. To ally to the general principles for user interaction in the DSUI we define these operations with respect to design objects rather than at the datatype level. For the *clear* operation on a specific port this means that if this port is colored with data of a design object, all data belonging to this design object is removed from the colored flow. The *assignment* of a design object to the design flow colors all ports and channels for which data exists within that design object. A *browsing* operation on a port returns all design objects that may color this port.

In chapter 5 we described the derivation of a maximal colored flow starting from a user-assigned data entity. The Nelsis-counterpart of a user-assigned data entity is a *user-assigned design object.* Starting from this design object, all design objects that are related can be derived and displayed in the colored flow until no more design objects can be displayed. One of the heuristics for choosing between different related design objects presented in chapter 5 is the determination of the degree of 'relatedness'. A refinement on this heuristic could be to take the other inter-design object relationships maintained by the Nelsis CAD framework, such as version relationships, hierarchical relationships and equivalence relationships into consideration for the determination of the relatedness. If one could configure which inter-design object relationships should be taken into account for which parts of the design flow, this would yield a powerful mechanism to further tune the behavior of the user interface to a specific design application. Such a 'flow coloring hint' could be configured using a specific type of design object port relationship.

A completely different aspect that needs to be addressed when applying flow coloring concepts in a design system user interface is how to cope with changes in the state of the design process. In the course of a design process the run-time information as administered in the meta data changes continually. For instance, the execution of a design tool extends the run-time information with a tool run, a number of activity runs and possibly multiple design and data transactions. A design flow user interface that displays the state of the design process in a colored flow, must respond in an intelligent way to these changes in the run-time information. It should adjust the colored flow displayed in such a way that important changes are shown and unimportant changes are left for what they are.

Conceptually, it is possible to derive a new colored flow upon any data transaction performed. However, since a design tool may perform many data transactions within one design transaction and since it may perform many design transactions within one activity run and many activity runs within one tool run, this would imply a potentially large number of updates of the colored flow, while designers are typically not interested in this detailed behavior of their design tools. On the other hand, updating a colored flow upon each completed tool run only would leave designers unaware of the completion of an activity run, which would nullify the advantages of using a fine-grain design flow management system. Therefore, the right level to update colored flows is the activity run level. This level is global enough to insulate designers from detailed changes in the design state and detailed enough to visualize the fine-grain progress of a tool at the level of activity runs.

Another situation in which the design flow user interface must update the colored flow displayed is when a design object is destroyed. In Nelsis, a design object may either be destroyed by one of the design tools within the scope of an activity run, or by a designer manually. In either case the design flow user interface should clear the ports and channels that were colored with this particular design object.

In a multi-user multi-tasking design environment each designer may have its own design flow user interface displaying a particular colored flow. Since designers do not like their colored flows to be updated upon design actions of their fellow-designers if this has nothing to do with their own work, the individual design flow user interfaces should update their colored flow under certain conditions only. A satisfying set of conditions for this is to display a new activity run only if there is at least one design object involved in that run that is also displayed in the colored flow, or if the run has been performed by the owner of the colored flow. Obviously, if a design object is destroyed this leads to a recalculation of the colored flow only if it is currently displayed. Since these rules are based on heuristics, designers should be able to explicitly request the design flow user interface to update its colored flow or to temporarily disable the update facility.

6.4.2 Design Scheduling in Nelsis

The general information model for design schedules (figure 5.28) addresses so-called scheduled data entities. In the Nelsis design flow model this could be extended to address scheduled design objects and scheduled data of a certain datatype. However, the question is in how much detail a design schedule should 'predict the future'. Is it really necessary to administer the expected data access behavior of scheduled activities in a design schedule at the datatype level, or is it sufficient to describe the access behavior at the design object level only?

To answer this question, we must keep in mind that in Nelsis design tools are invoked by referring to the design objects on which they should operate and not by referring to the specific data within these design objects. This observation implies that for the invocation of an activity of a design tool in the scope of a design schedule, the design flow system does not need to know which data within a design object is to be accessed. It suffices to administer the scheduled design objects for scheduled activities. The refined information model is shown in figure 6.11. The object type *ScheduledDesignTransaction* describes the access of an activity at the design object level.

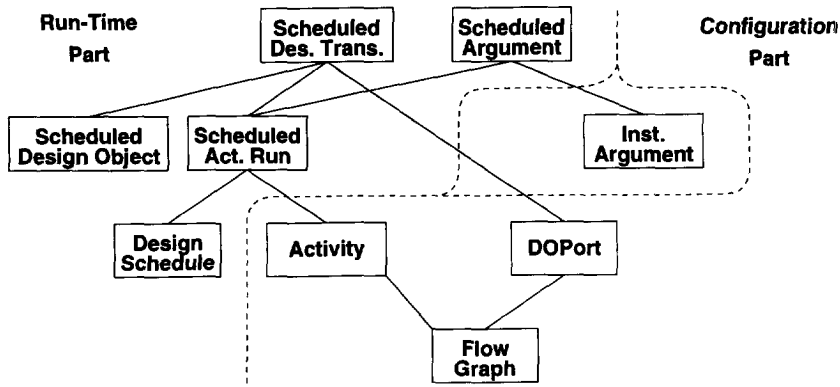


Figure 6.11. Modeling design schedules in Nelsis.

This refinement does not really affect the way design schedules are executed and represented. The coloring of a Nelsis design flow with design schedule information can be performed in almost the same way as the coloring of a Nelsis design flow with run time information, except that in this case activities are colored with scheduled activity runs and ports and channels are colored with scheduled design objects.

6.5 Changing Design Flow Configurations

Until now we supposed a design flow configuration to be stable during the design process. However, in reality there may be a need to change the design flow configuration in the course of the design process. This may happen for instance when new design tools or new versions of existing design tools become available or when the design process gets into a new phase that brings about new design requirements or new design constraints. Taking into account that design tools are continuously being developed and that in an explorative design process the tool set may not be known completely when a design flow is being defined, the design flow system should support the redefinition of design flows.

Since the design state as administered in the run-time information refers to design flow primitives such as activities and ports, changing the design flow configuration without any further precautions may cause the run-time information to reflect an incorrect design state. For instance, the addition of a streamtype to a datatype, may imply that an existing design object is assumed to contain a stream of the added streamtype while in fact this is not the case. To prevent designers from being advised incorrectly, the design flow system should handle changes to the design flow configuration with care.

Upon a change in the design flow configuration it may be impossible to adapt the run-time information in such a way that the old design state is administered in the new design flow configuration. For instance, in the example of the extended datatype definition, data transactions may no longer refer to a datatype that contains the correct streamtype definition. Therefore the best the design flow system can do is to refrain from preserving the old design state, and to *mark* run-time information that is no longer consistent with the new design flow configuration. An advantage of this mechanism is that designers can be warned for an incorrect design state, while they can still use the old design data if they like.

Below, we briefly describe such a marking mechanism. It is very well possible to describe for each type of change which part(s) of the run-time information need to be marked. However, since changes to the design flow configuration tend to occur conjointly, we adopt a worst-case scenario for marking the run-time information. To ally to the general principle that users interact at the level of design objects, we let the marking of run-time information work its way to the design object level. We make a distinction between three different classes of changes, each with their own characteristics. This subdivision originates from the different parts of the design flow configuration and their use. We distinguish between:

1. Changes in the data definition part.
   This part describes the organization of data into viewtypes, datatypes and streamtypes. Changes in the data definition part of the design flow configuration are not likely to occur very frequently since different tools in a design environment must agree on a common data organization. However, it may happen that the introduction of a new tool makes it necessary to revise the data organization. Since the ports of a possibly large number of different activities of possibly different tools may refer to one and the same viewtype or datatype, a change in the data definition part may influence a large amount of run-time information.

   A change in the viewtype definition may possibly influence all design objects of that viewtype and a change in a datatype definition may possibly influence all data of that datatype in the different design objects. Therefore, a worst-case marking strategy for changes in the data definition part of the meta data is to mark all design objects for which there exists a design transaction on a viewtype that has been changed or a data transaction on a datatype that has been changed.

2. Changes in the tool description part.
   This part describes the definition of the tools, their activities and the ports of these activities. Examples of changes in the tool description part are the change of a port, the addition of an activity to a tool, or the addition of a tool. This type of changes is likely to occur if design tools are still under development. The amount of run-time information influenced by such a change is limited: only run-time information that pertains to activity runs of the changed tool is influenced.

   A worst-case mechanism for changes in the tool description part is to mark a design object if it is involved in a design transaction that refers to a design object port for which the definition has been changed, or for which there is a data transaction on a data port for which the definition has been changed.

3. Changes in the design flow part.
   This part comprises the definition of the design flow hierarchy and the definition of dependencies between instantiated flow graphs. With respect to a change in the hierarchical decomposition of the design flow we call into mind that for each activity run the instances of this activity are administered by the full path to the root of the flow graph hierarchy (section 4.3). Therefore, a change in the flow hierarchy should result in a marking of the activity instance runs involved. With respect to the definition of the dependencies things are different. If a dependency is changed so that an activity run that has been performed on some design object can no longer be performed with the new dependencies, this does not have to lead to a marking of the run-time information. Hence, activity runs, once performed, maintain their validity even if the dependencies are changed.

This marking strategy determines a very rough solution to the problem of changing design flow configurations. Many more refinements could be made, for instance to distinguish between more types of changes, or to use the number of markings of a design object as an indication of the uncertainty of its state. However, we think that the general strategy to mark design objects upon changes in the design flow configuration so that the end-user can be warned while the design data is still accessible, is a satisfying compromise between a more strict policy where such a change invalidates all design data that is affected and a more lenient policy where such a change does not affect the run-time information at all.

To facilitate the (re)definition of design flow configurations, the Nelsis design flow system offers a special purpose graphical design flow editor, the *FL*ow *U*ser *I*nterface *D*esigner (FLUID) [Nelsis93]. This framework tool is the appropriate place to implement a marking mechanism as described above.

6.6 Design Flow Management in the Nelsis Architecture

In this section we examine how the design flow system can be integrated into the architecture of the Nelsis CAD framework with the constraint that it preserves the open and efficient character of this system. First we locate the different components of the Nelsis design flow system into the component architecture of the Nelsis CAD framework, then we describe the distribution of design flow functionality among the Nelsis processes and finally we describe how the general notification mechanism in Nelsis can be exploited in the design flow system to keep designers well-informed and to handle changes in the design flow configuration correctly.

6.6.1 Implementation in the Nelsis Component Architecture

The component architecture of a system describes its different components, the dependencies between these components and their interfaces. The component architecture of the Nelsis CAD framework, as shown in figure 6.12, conforms to the general CAD framework architecture as presented in chapter 2. There is a clear distinction between the usually greater part of design data, organized into design objects, and the usually smaller meta data, which is stored in an efficient semantic data base organized according to an OTO-D data schema. The framework kernel comprises a set of services such as concurrency control, version management, hierarchy management, access control, etc. Tools communicate with the Nelsis CAD framework via the dmi, which is a procedural interface that offers functions for accessing design data as described in section 6.2.2 (initialize / terminate tool run, open / close project, check out / check in design object, open / close stream) as well as for managing the hierarchy of a design, managing versions of design objects, adding and inspecting equivalence relationships between design objects and more low level functions for querying any information about the design process from the meta data. The DSUI is implemented as a framework tool, which communicates with the CAD framework kernel through the same dmi as the other tools, with the exception that it does not access design data. It restricts itself to the visualization of the meta data.

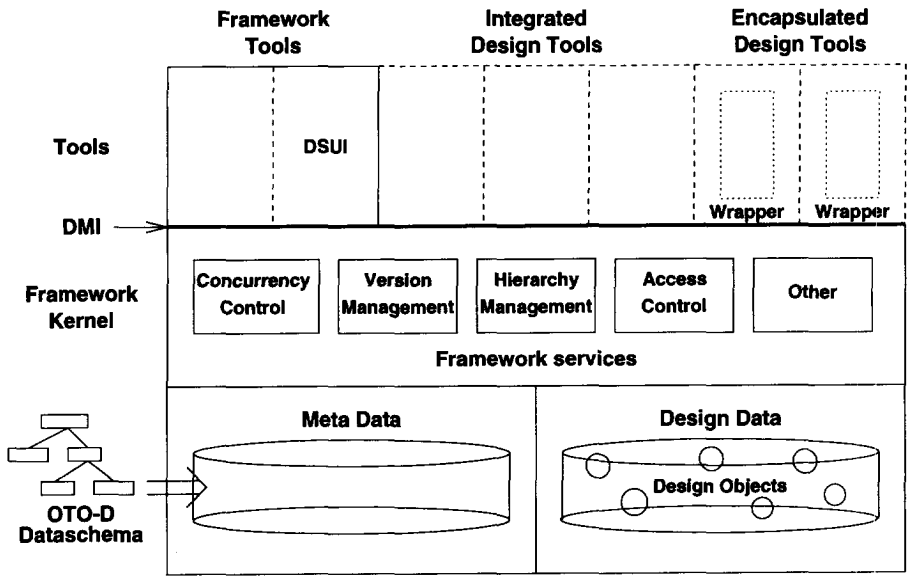


Figure 6.12. The Nelsis component architecture.

To extend the Nelsis CAD framework with design flow management, it is important to integrate the different components of the design flow system in the right way. In chapter 2 we already stated that a design flow system should be spread over two distinct design flow modules: the design flow kernel, which performs those design flow functions that should be performed centrally, and the design flow user interface, which offers design flow functionality that can be used by designers if they like, such as flow-based user interaction.

Functions that should be performed in the design flow kernel of Nelsis are the activity recognition process, the stream to data mapping and the enforcement of design constraints configured in the design flow. These functions are kernel design flow functions since they must be performed for any tool run of any tool invoked by any designer. If one of these functions is not implemented in the design flow kernel but in the design flow user interface, runs of design tools that are not invoked via the design flow user interface are not recognized, administered or checked correctly. A design flow architecture where this critical design flow functionality is performed centrally in the kernel of the CAD framework guarantees that even if designers choose not to use the design flow user interface to invoke their design tools, the activities of the design tools are recognized and administered and the constraints configured in the design flow can be enforced, if so required.

The presentation of the design state in a colored flow and the interaction with the end users via flow coloring operations and browsing operations and the invocation of design tools from a colored flow are functions that should be implemented in the design flow user interface. In Nelsis, we choose to integrate the design flow user interface in the main user interface of Nelsis, the DSUI. This contributes to a more complete integration of design flow management with other services of the Nelsis CAD framework. For instance, it facilitates the simultaneous presentation of a maximal colored flow derived from a user-assigned design object and the hierarchical decomposition of the same design object.

Figure 6.13 shows the component architecture of the Nelsis CAD framework with the different (shaded) design flow components. The OTO-D data schema, which specifies the organization of the meta data, is extended with the configuration part and the run-time part of the information models presented in the previous chapters. The framework kernel is extended with a design flow kernel and the DSUI with a design flow user interface component. The design flow configuration editor, FLUID, is implemented as a framework tool.

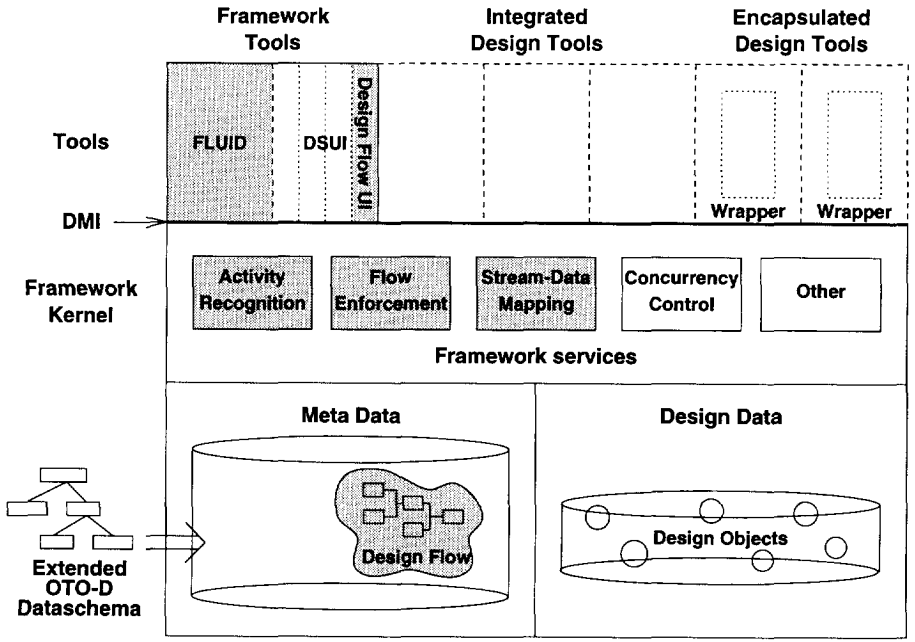


Figure 6.13. Design flow management in the Nelsis component architecture.

6.6.2 Implementation in the Nelsis Process Architecture

Since the Unix® operating system has proved to be the common operating platform for the more complex design systems, it has been chosen as the basis for implementing the Nelsis CAD framework. Not only does Unix support the concurrent execution of multiple programs, known as *processes*, it also offers extensive facilities for communication among different processes possibly running on different machines, known as *inter-process communication (IPC)*.

In Nelsis, each tool runs as an individual Unix process. Framework functionality is spread over the tool processes and a number of server processes. Framework functionality that can better be executed as part of the tool processes for some reason, is linked to the tools. All other framework functionality is spread over a number of processes in such a way that for each active design project there is exactly one *project server* process. A major advantage of this distribution of framework functionality over tool processes and project server processes is that the system is *scalable* in the sense that an increase in the number of design activities (running tools) or in the amount of design data (design projects) implies a proportional increase in the framework capacity.

Tool processes operating in a specific design project communicate with the project server process for that project through IPC. Since a tool may actually operate simultaneously in different design projects, it may communicate with different project server processes at the same time. Important reasons for performing framework functionality into a tool process instead of into a project server process, are for example an easy management of tool-specific data structures or a minimization in the load of the IPC connections between the tool processes and the project server processes.

Figure 6.14 shows the Nelsis process architecture in an intuitive way. This visualization method is borrowed from [Wolf94a]. Circles denote unix processes, solid lines connecting circles indicate IPC connections, dashed lines connecting processes and disk units indicate that these processes perform data access and triangles describe nelsis design projects. A dashed line within a circle visualizes the border between the design tool code and the part of the CAD framework that is

® UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a Trademark of X/Open Company Limited in the UK and other countries.

linked to the design tool. This figure shows two tools running as individual processes. Tool 1 operates in project *A* only, tool 2 operates in project *A* and *B*. The project server for project *A* communicates with both tool processes.

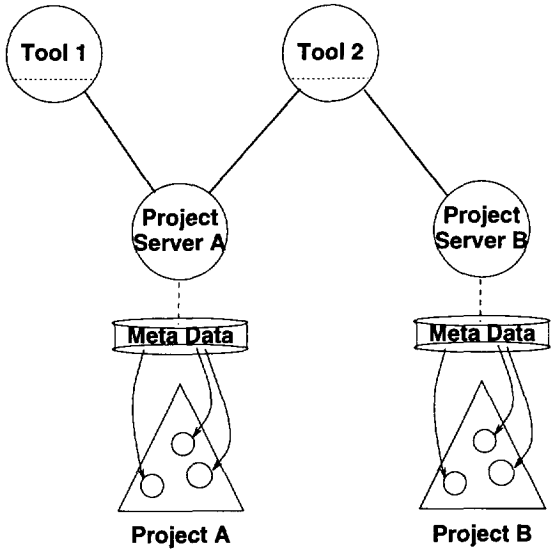


Figure 6.14. The Nelsis process architecture.

Since the distribution of design flow functionality among Unix processes is crucial for the net-performance of the resulting system, we describe which design flow functions can best be linked to the tools and which can best be implemented in the project server. Important criteria for this are the amount of meta data that has to be inspected and the load of the IPC connections.

For recognizing the activity runs of a tool from its data accesses, the activities of this specific tool need to be inspected only. Since this is only a small part of the design flow configuration and since this information is tool-specific, it can best be implemented in that part of the CAD framework that is linked to the tools.

Since the mapping of streamtypes onto datatypes is not tool-specific at all, it seems probable to integrate the stream to data mapping function into the project server. However, a disadvantage of this approach is that during activity recognition each tool process has to request this information from the project server process, which would seriously increase the load of the IPC connections. Therefore, and since the datatype configuration takes only a small part of the design flow configuration, we think it is better to perform the stream to data mapping function in the tool processes.

In contrast to activity recognition and stream to data mapping, the enforcement of design constraints may take a large part of the design flow configuration to be inspected, while the result is a simple yes / no answer. These characteristics make that this function can better not be linked to the design tools. Instead, if the project server loads the complete design flow configuration for a design project in core, it can efficiently answer requests about design constraints from multiple tool clients, without sending large parts of information via the IPC connections. In addition, since a project server survives multiple tool runs, this choice reduces the startup-time for subsequent tool runs.

The DSUI and FLUID are special processes in the sense that, to derive colored flows to perform browsing operations, to perform flow coloring operations, and to allow the editing of the design flow configuration, they must have the complete design flow configuration at their disposal. Therefore, these flow-specific framework tools must load the design flow configuration in core.

The overall process architecture of the Nelsis CAD framework, extended with design flow management functionality, is shown in figure 6.15. The shaded components represent information or process functionality that is part of the design flow system. The shaded right parts of the DSUI processes represent the flow coloring mechanisms as implemented in the design flow user interface. The graphical design flow editor FLUID runs as an individual process. To avoid conflicting editing sessions for one and the same design flow configuration we allow at most one FLUID process per project.

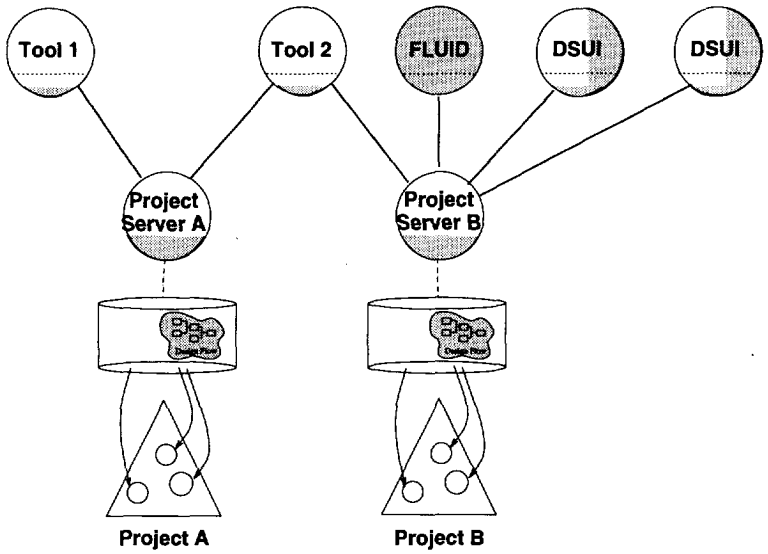


Figure 6.15. Design flow management in the Nelsis process architecture.

6.6.3 Notification

In section 6.4.1 we described how a design flow user interface should update a colored flow upon changes in the run-time information and in section 6.5 we addressed the implications of modifying the design flow configuration. In both situations one particular process changes part of the meta data. We did not describe how other processes than the one that causes this change find out about such a change. This is particularly important for a design flow user interface process showing a colored flow that is based on information administered in the meta data. To correctly show the design state to the designers the design flow user interface processes must somehow find out about changes in the meta data. A solution where these processes inspect the meta data again and again (polling) to see whether something has changed is objectionable for reasons of efficiency and also unnecessary, since the Nelsis CAD framework offers a generic *notification* mechanism that can be used.

The Nelsis notification mechanism exploits the existing IPC connections between a project server process and its clients (the tool processes) to broadcast information about the occurrence of events of predefined types. To reduce the load of the IPC connections and to prevent clients to be notified of all possible events, each project server process sends a notification of the occurrence of a certain type of event to clients that explicitly registered their interest in this type of

event only. An event may be triggered by one of the clients of the project server process or by the project server process itself. Each receiving client independently decides how to respond. Since events may in fact be triggered by a specific set of database transactions, the Nelsis notification mechanism can be used to update colored flows displayed by the different design flow user interface processes upon changes in the design state and to handle changes in a design flow configuration of a project correctly.

As described in section 6.4.1 each design flow user interface process may have to update its colored flow upon the completion of an activity run or the removal of a design object. This implies that the Nelsis notification mechanism should be extended with two types of events: one for the successful termination of an activity run and one for the removal of a design object. The design flow user interface processes register their interest in these two types of events, which are triggered by the part of the design flow system that is linked to the design tools. Figure 6.16 shows the notification of the design flow user interface processes upon the completion of an activity run in the Nelsis process architecture. A dashed arrow near an IPC link indicates that a notification message is sent across this link.

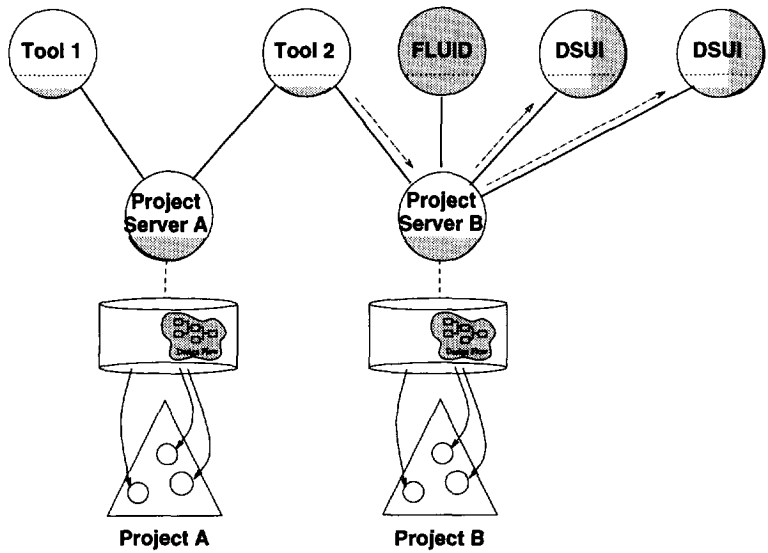


Figure 6.16. Notification of the termination of an activity run of tool 2.

Another use of the notification mechanism is for handling changes in the design flow configuration as described in section 6.5. For this, we introduce a new type of event that indicates that a design flow configuration has been changed. Since all tool processes have some part of the design flow configuration in core (for

instance the definition of the activities for a tool), conceptually all clients of the project server should be interested in events of this type. However, since a change of the design flow configuration is not expected to occur very frequently and since a change in the design flow configuration in the middle of a tool run could be the cause of strange effects, we introduce a constraint that FLUID may update a design flow configuration only if no design tools are active on this project. This constraint implies that only the design flow user interface processes have to be notified. Figure 6.17 shows the notification of the design flow user interface processes upon a change in the design flow configuration caused by the FLUID process. Note that in this example tool 2 does not communicate with the project server process *B*, which is in line with the constraint mentioned above.

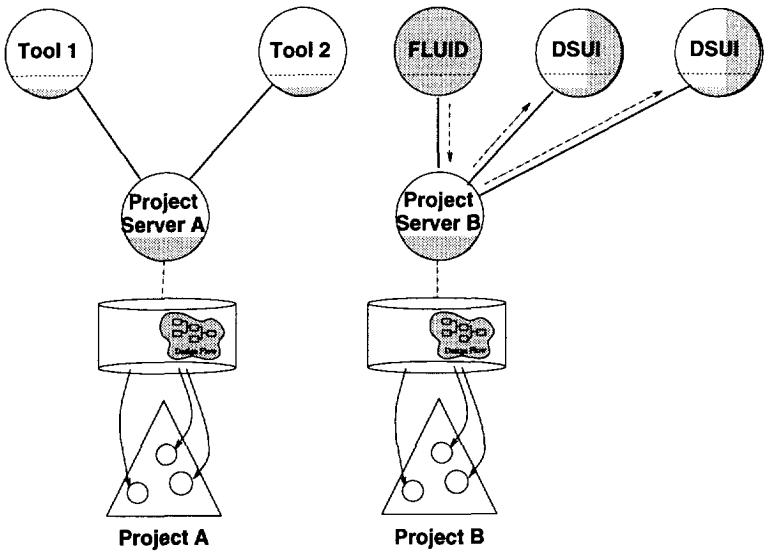


Figure 6.17. Notification of a change in the design flow of project B.

One common pitfall that we have to get around when designing a communication protocol between concurrently executing processes like the notification mechanism presented in this section, is the occurrence of deadlock (a number of processes is waiting for each others notifications) or lifelock (a notification causes a never-ending domino effect of consecutive notifications). The absence of deadlock is guaranteed, since none of the processes is actually waiting for a notification request. The notifications described are informative only. If they stay out for some reason the design flow user interface processes just keep working. Lifelock cannot occur since for all three types of events holds that the notification is sent to the DSUI processes only, which do not pass the notification any further.

6.7 Related Issues

Design flow management is only one of the many services of a CAD framework. For a successful integration of a design flow system into a CAD framework it is important that a design flow system does also cooperate smoothly with the other framework services. In this section we address the interplay of the Nelsis design flow system with two other CAD framework services: library management and access control.

6.7.1 Design Flow Management and Library Management

To enable a designer to use design data that is actually residing in some other design project than the one he is working in, the Nelsis CAD framework offers a mechanism to explicitly *import* a number of design objects from one project into another. Instead of physically copying the design data, the import mechanism administers a reference in the project the designer is working in, the *local* project, to the project in which the design object resides, the *library* project. A library project is not different from any other project. This import mechanism can be used to define a design environment with a number of library projects containing design data that is to be used in many other projects and possibly by many different designers. We capture these facilities under the name *library management.*

To let the Nelsis design flow system cooperate with the import facility, it should be possible to show imported design objects in a colored flow and to invoke design activities on them. For this, the state of an imported design object as administered in the meta data of the library project should be consulted. However, since the local design project and the library design project may have completely different design flow configurations, this may cause problems. The state of an imported design object as administered in the meta data of the library project may refer to design flow primitives that are absent in the design flow configuration of the local project. In this case it is impossible to show the state of an imported design object in a colored flow in the local project.

This problem can be attacked in a number of ways. One possibility is to express the state of a design object in project-independent terms such as streams. However, to represent a design object in a colored flow in the local project it is still necessary to calculate a project-dependent state in terms of the datatypes in the local project. Since the streamtype to datatype mapping in the local project may differ from the streamtype to datatype mapping in the library project, it may still be impossible to correctly show the design state of the imported design objects in the local project.

Another method is to refrain from the determination of a detailed design state and to restrict the state of an imported design object to its viewtype only. Although this method makes it impossible to show the state of an imported design object in detail, it has the advantage that the import mechanism can be supported in a clear and understandable way. The only constraint for importing design objects is that the viewtype of the design object in the library project also appears in the local project. In the rest of this section we suppose that this method is used.

To show imported design objects in a colored flow and to allow design activities to be invoked on them from a colored flow, the design flow configuration of the local project should be extended with a number of activities that explicitly define the importation of a design object from a library project into the local project. This leads to a so called *import activity* for each viewtype for which design objects are to be imported.

Figure 6.18 shows the design flow of figure 6.10 extended with two import activities, one for importing design objects of the viewtype of the output port of activity *a* and one for importing design objects of the viewtype of the output port of activity *c*. The dependencies in the design flow show that design objects imported by activity *import*1 can be accessed by activities *a* and *b* and design objects imported by activity *import*2 can be accessed by activity *d*. The coloring of the design flow shows, among other things, that a design object *do*6 has been imported by activity *import*2 and that this imported design object has been accessed in activity run *ar*8 of activity *d*.

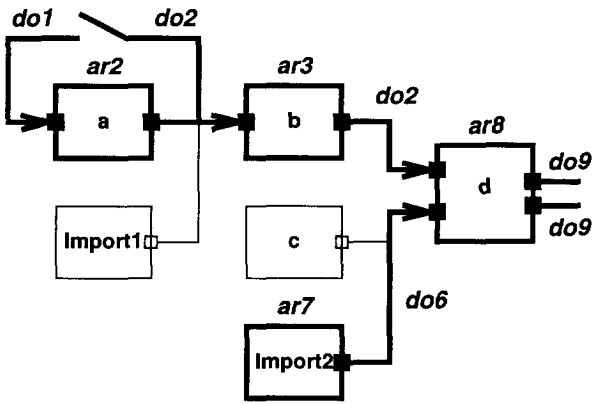


Figure 6.18. The visualization of an imported design object (*do*6) in a colored flow.

6.7.2 Design Flow Management and Access Control

The need to control the access to different resources in a design environment varies from application domain to application domain. For instance, in a research environment it may be unwise to restrict designers in their design freedom where in a production environment it may be desirable to do so. For design applications that need a more strict design philosophy and in which an access control mechanism is active, it may be useful to extend the access control mechanisms to cooperate with the design flow mechanisms.

A straightforward extension of an access control mechanism in a design flow system is to allow access permissions to be assigned to flow graphs. In this way, it can be configured which designers or group of designers may execute or inspect a specific flow graph. Without going in too much detail, we note that in general an access control mechanism allows *designers* to be organized in *groups* with a *role* (such as project leader, design engineer etc.) assigned to their group *member*ship (see for instance [CFIfar93]). Designers may be a member of many different groups and may play a different role in each group. To allow access permissions to be assigned to designers playing a specific role in a group, we create the opportunity to hand out *privileges* to roles, where a privilege is the right to access a *flowgraph* in a certain way (for instance, to execute the flow graph or to open it).

Figure 6.19 shows an information model for such a general access control mechanism. It shows that a *Designer* is a *Member* of a *Team* and has a certain *Role* in that *Team*. A *Permission* assigns a *Privilege* to a certain *Role* and a *Privilege* is the right to access a *FlowGraph* in a certain way, defined by the *Type* attribute.

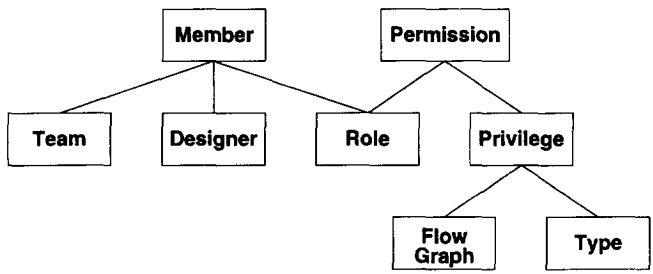


Figure 6.19. Information model for assigning access rights to flow graphs.

A more detailed description of access control in CAD frameworks (including the integration with a design flow system) can be found in [Hoeven94].

6.8 Examples

The Nelsis design flow system has been used in a number of different application domains to model the design process in a design flow and to take advantage of flow based user interaction. In this section we describe a number of design flows that have been configured in different places and the lessons we learned from modeling the characteristics of the different tools and their interelationships.

6.8.1 The Nelsis IC Design System

The first design environment in which the Nelsis design flow system has been tested is the Nelsis IC design system[3]. This design system consists of a set of tools, developed at Delft University of Technology and other places, for layout design and verification, circuit design and simulation and place & route (see also[Dewilde86]).

The tools of the Nelsis IC design system were already integrated in the Nelsis CAD framework before it offered any capabilities for design flow management. Due to the clearly defined interface between the tools and the CAD framework (the earlier mentioned dmi), the development of the tools could be decoupled from the development of the CAD framework. In fact, even the addition of design flow management to the Nelsis CAD framework caused no major changes to the standardized interface, so that the tools of the Nelsis IC design system could take advantage of design flow management without any reprogramming.

It is interesting to see that before the Nelsis CAD framework offered any design flow management, there was already an informal idea of a 'design flow' in the Nelsis IC design system. However, this design flow was not formalized in any way. It existed as a picture in the system documentation and the designers had to keep this picture in mind during their design activities or look it up every time they needed it. Figure 6.20 shows this informal design flow. A number of design tools (*dali, kic, cmsk, xcmsk, cldm, xldm* etc.) are positioned around a central database containing data of one of three different viewtypes (*layout, floorplan* and

3. It may be confusing that the name Nelsis is used for the domain neutral CAD framework as well as for this domain specific IC design environment. The reason for this is that the initial development of both software systems started in the same 'Nelsis' project. To exclude any misunderstandings in this section we will write 'the Nelsis IC design system' for the set of domain specific IC design tools and 'the Nelsis CAD framework' for the domain neutral software infrastructure.

circuit). Arrows indicate the dependencies between the database and the design tools and between design tools and design data that is *not* stored within the central database (such as a *CMSK, LDM, FUNC* or *SLS* description).

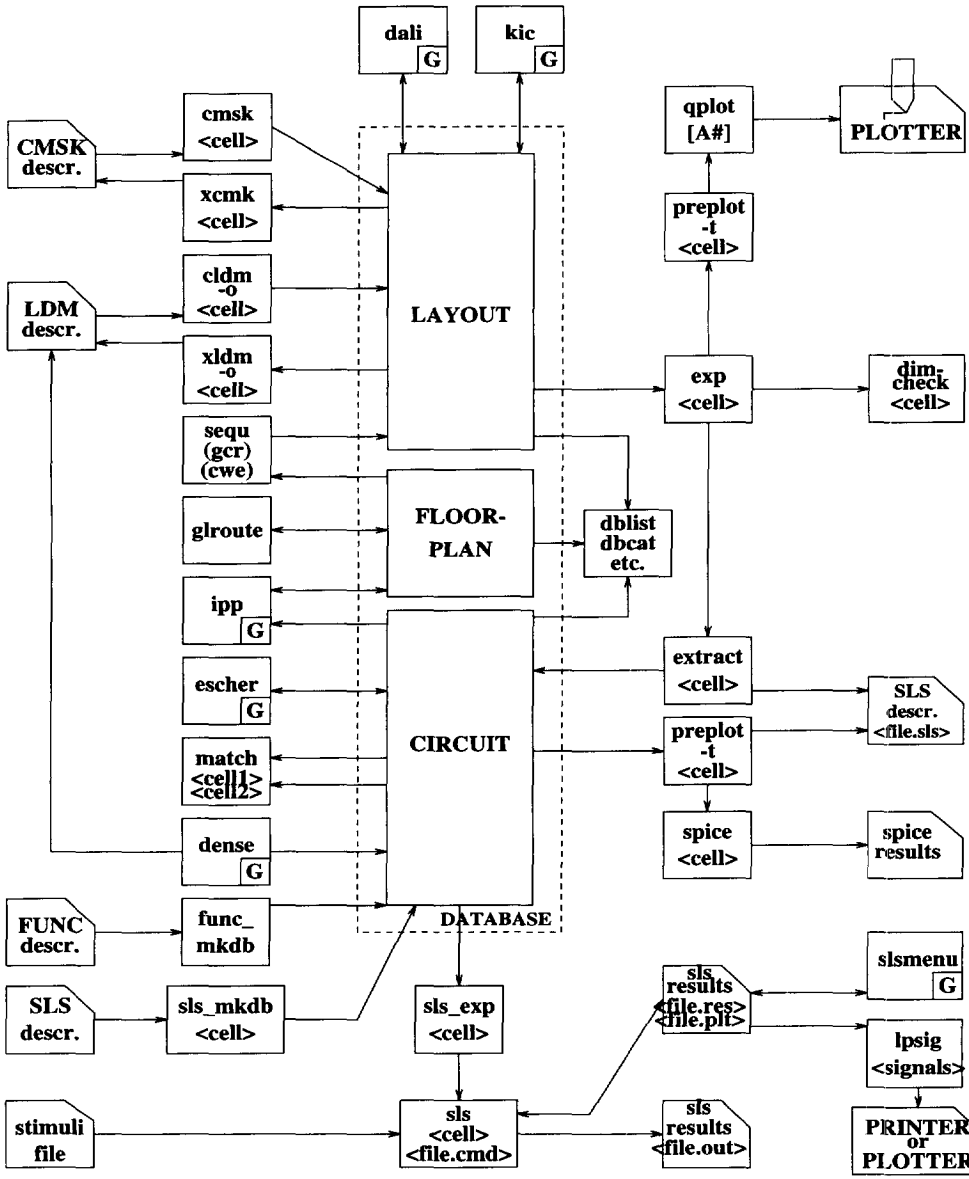


Figure 6.20. The informal design flow of the Nelsis IC design system.

With the advent of design flow management in the Nelsis CAD framework, the informal design flow could grow from an abstract picture in the designers' minds into a well-defined formal description of the dependencies and the constraints in the design process managed by the CAD framework. The presentation of a colored flow has become an important mechanism for designer interaction in the Nelsis IC design system. Layout and circuit design objects are shown in the context of the tools that generated or modified these design objects and tools can be invoked from this colored flow. Figure 6.21 shows a colored flow as displayed in the design flow user interface for the Nelsis IC design environment. It shows flow graphs, ports, channels, switches, and 'balloons' indicating which design objects color a channel. Flow graphs with double lines are compound flow graphs.

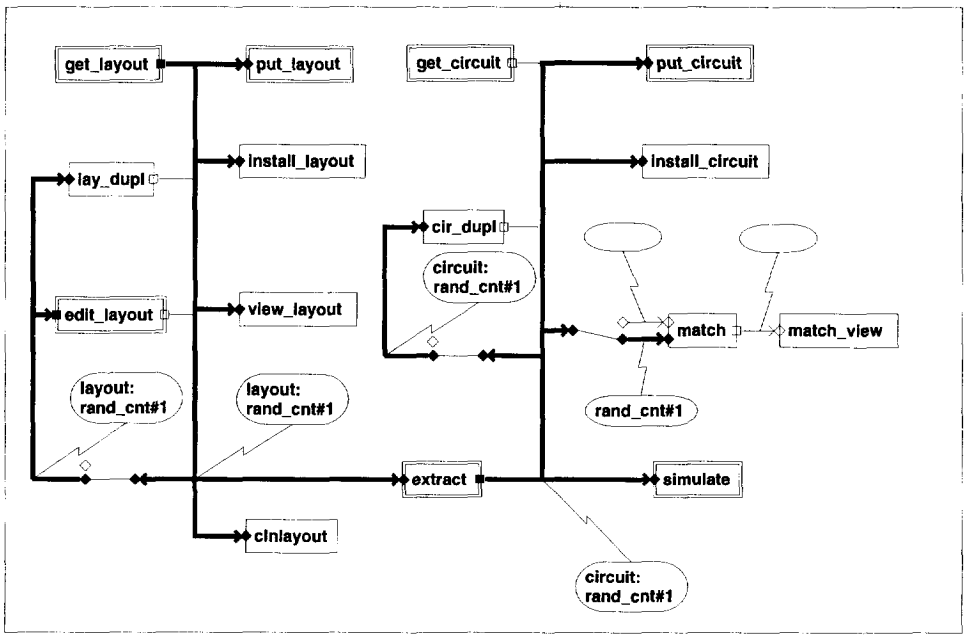


Figure 6.21. A colored flow for the Nelsis IC design system.

Some of the tools of the informal design flow, like *extract* and *match*, occur in almost the same way in the formal design flow. Others, like *cldm*, *dali* and *sls*, are hidden in compound flow graphs. Figure 6.22 shows the inside of the compound flow graph *simulate*. Note the similarity of this picture with the corresponding part of the informal design flow. The tools *sls_exp*, *edit_stim*, *sls* and the simulation result viewer *simeye* in figure 6.22 are the formal equivalents of the symbols *sls_exp*, *stimuli file*, *sls* and *lpsig* in figure 6.20.

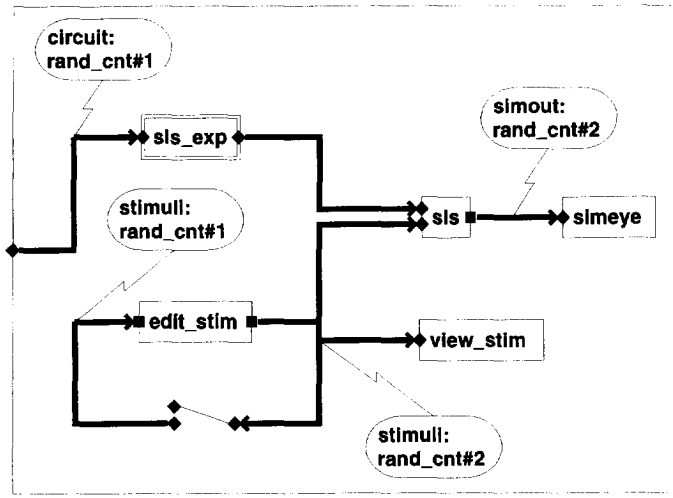


Figure 6.22. The compound flow graph *simulate*.

One of the most striking experiences that came up when building design flows for the Nelsis IC design system, was that for some tools the definition of the activities could be quite difficult. Especially for earlier versions of the Nelsis design flow system, which contained few flexible constructs for describing the activities of design tools (such as optional ports), the number of activities could be more than desirable from a user's point of view. Later versions of the Nelsis design flow system showed that an increase in the flexibility of activity definition decreases the number of activities that need to be configured for a design tool. The general feeling is that the use of a design flow system provides designers with a clear view on the design environment, which contributes to a more efficient design process.

6.8.2 An Analog Simulation Environment

Another test of the Nelsis design flow system has been performed at Philips Research Laboratories. An analog simulation environment containing design tools for schematic editing, waveform editing and display, and netlist expansion and simulation has been described in a design flow. One of the tools in this simulation environment was a fairly complex interactive editor, which was *not* developed at Philips and which therefore had to be encapsulated in Nelsis. This project showed that using smart encapsulation scripts the state of the design process can be shown at a fine-grain level, even if design tools are encapsulated instead of integrated into the underlying CAD framework.

6.8.3 A Simulation Environment for Medical Applications

Another effort performed at Philips Research Laboratories was the use of the Nelsis design flow system for the construction of the Image Generating Simulations environment (IGS). This project has been described in [Lepoeter93]. Figure 6.23 shows a colored flow for this simulation environment. To perform a simulation, a configuration description as well as a settings description has to be chosen. There is a dedicated editor for both types of descriptions. The simulation process is split up into three individual steps: the *ISS.XP* simulation, the *ISS.ID* simulation and the *ISS.IP* simulation step. After each step, the results can be viewed and analyzed.

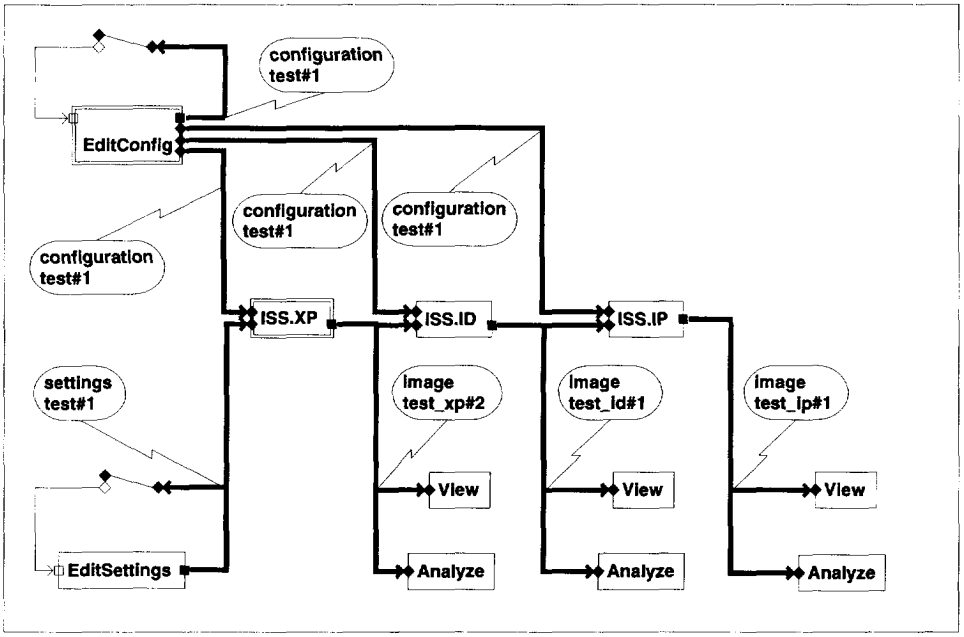


Figure 6.23. A colored design flow for the Image Generating Simulations environment.

The *ISS.XP* simulation actually takes a number of smaller steps to be performed, which are explicitly modeled within the compound flow graph *ISS.XP*. The internals of this flow graph are shown in figure 6.24.

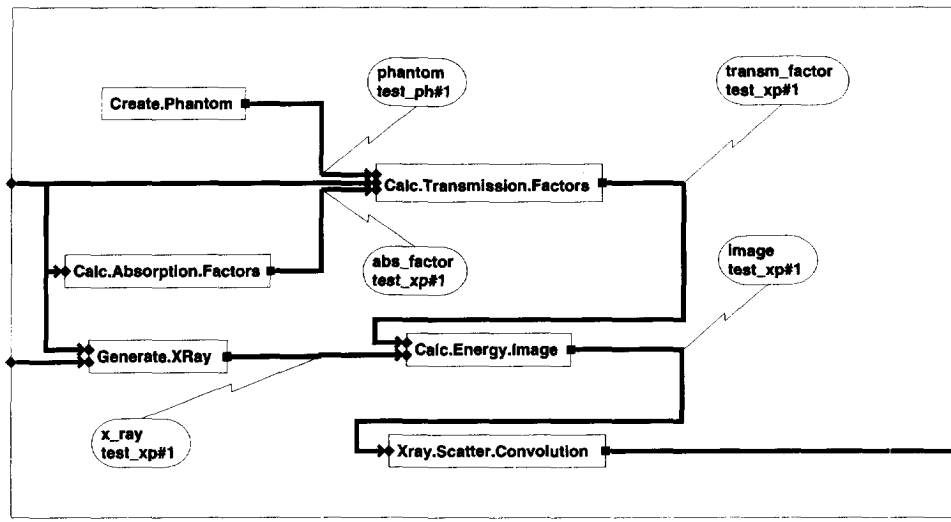


Figure 6.24. The internals of the compound flow graph *ISS.XP*.

An important observation that resulted from this project was that the heuristics used by the design flow user interface to derive a maximal colored flow play an important role for the perception of the design flow system. When a former version of the design flow user interface, which did not use the heuristics described in section 5.3.6, was used on this particular design flow, problems showed up when deriving a maximal colored flow under certain specific conditions. For instance, if starting from an uncolored flow the user assigns a design object to the output port of flow graph *ISS.ID* the resulting maximal colored flow depends on the direction in which the design flow user interface starts searching for related design objects. If it starts to search for design objects related via flow graph *ISS.IP*, it finds a different colored flow than if it starts to search for design objects related via flow graph *ISS.ID*. In both cases, the result is a colored flow that satisfies the flow coloring constraints as presented in chapter 5, but from the viewpoint of the designer, one of these colored flows is more desirable than the other. The fundamental reason for this problem is that a colored flow can show only part of the design state and therefore the design flow user interface must sometimes make an intelligent guess about which information to show. In the former version of the design flow user interface this guess was not smart enough. This experience contributed to a refinement of the heuristics, for deriving a maximal colored flow and the possibility to fine-tune the coloring algorithm for a particular design flow using port relationships.

6.8.4 The Automatic Nelsis Test Environment

The Automatic Nelsis Test (ANT) environment consists of a set of tools aimed at the design and management of test cases for software products and their automatic execution [Schot92]. In 1993 a design flow for the ANT system was designed [Bartels94]. This design flow shows the organization of so-called *test objects* in different viewtypes and the operations that can be performed on these test objects, such as executing a test or reporting the test result.

One of the difficulties that came out of this project was that tools that are so generic that they can execute on any stream in any viewtype are difficult to configure. In the example design flow this caused a tremendous number of almost similar activities for one and the same design tool and a huge number of data dependencies. This experience contributed to the introduction of a number of constructs for flexible tool characterization in the tool model, such as optional streams, optional datatypes and optional design object ports. These constructs can be used to describe less important tool characteristics less precise, while retaining the exact description of essential tool characteristics. This reduces the number of activities for a tool and with that the number of dependencies considerably.

Using these new constructs, in 1995, another design flow has been defined for the ANT system [Disselkoen95]. Figure 6.25 shows part of this design flow. The design flow shows the different steps to be followed for the definition of a test. In this colored flow these steps have already been performed and a test object has been generated.

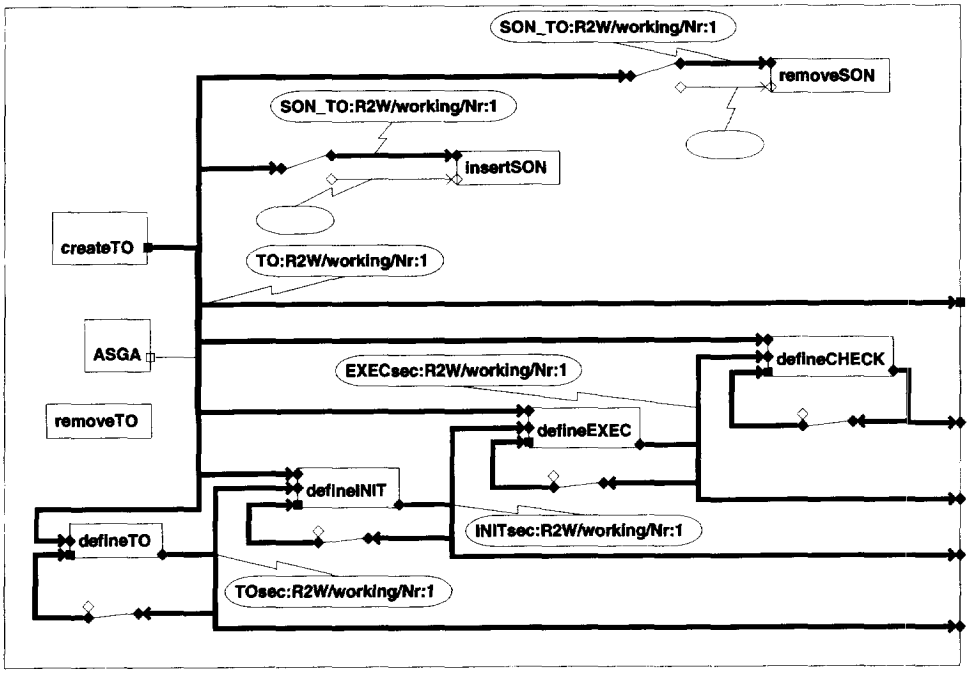


Figure 6.25. A colored flow for the Automatic Nelsis Test Environment.

6.8.5 The HiFi Design System

The HiFi (*H*ierarchical *I*(n)tera(c)tive *F*lowgraph *I*ntegration) design system [Held93] has been developed for the design of parallel algorithms and architectures for signal processing. The HiFi design system uses the services of the Nelsis CAD framework for managing its design data as well as to take advantage of design flow management [Held94]. One of the many different design flows that have been defined for the HiFi design system during its still continuing development is shown in figure 6.26. This figure shows the compound flow graph *simhifi*, which contains a number of activities for the creation, parsing and editing of *hifi* descriptions, activities for the definition of *hifisignals*, an editor and a compiler for design objects of the *hififunc* viewtype, an editor for *hificommand* descriptions and a simulator *simulatehifi* that takes design objects of all these viewtypes as input. This picture shows only a part of the 45 flowgraphs that were used to model the complete HiFi design system.

The use of the Nelsis design flow system in HiFi has been interesting as it successfully combines the use of design flow management with the use of other

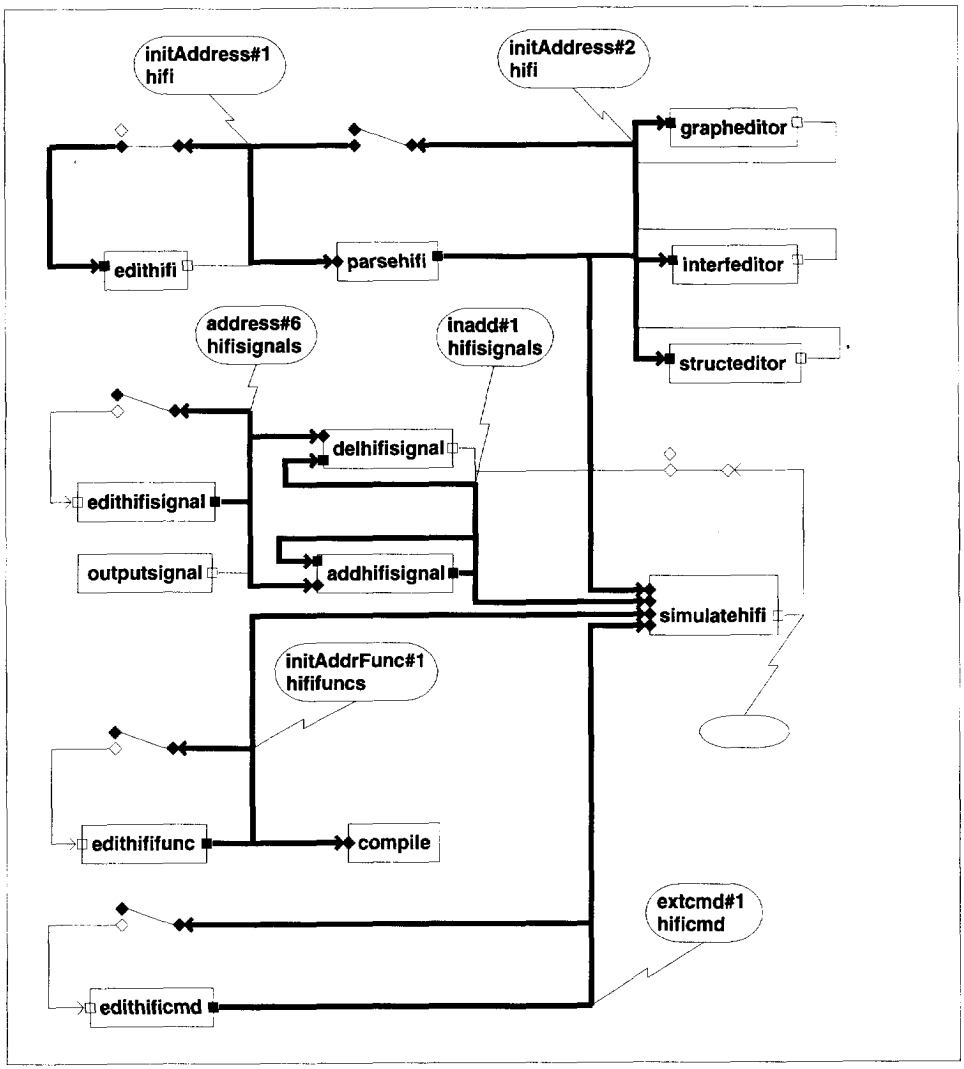


Figure 6.26. A colored flow for the HiFi design system.

features of the Nelsis CAD framework. For instance, in the HiFi design environment, which comprises integrated as well as encapsulated design tools, design data is organized in several different viewtypes. For some of these viewtypes the design objects may be decomposed in a (one-level) design hierarchy. The Nelsis design flow system allows tools for manipulating the hierarchical decomposition to be explicitly configured in the design flow. For instance, the activities *addhifisignal* and *delhifisignal* actually add and remove a

child hifisignal design object to or from its parent design object. Another aspect in which the HiFi integration project is special is the heavy use of the TES-based tool control description part of the Nelsis tool model and its accompanying graphical browser. Many HiFi design tools have a large number of command line parameters. Their invocation characteristics have been described in a TES-description to be used by the tool invocation browser of the design flow user interface to build a graphical tool invocation form, tuned for the tool in question.

6.8.6 The Cathedral-2nd High Level Synthesis System and the Chess Retargetable Code Generation Environment

The Cathedral-2nd high level synthesis system [Lanneer90], developed at Imec, Belgium, comprises a set of tools for automatic synthesis of architectures for real-time signal processing (DSP). These tools have been encapsulated in the Nelsis CAD framework in just a few weeks. Since most of the tools were encapsulated using almost similar wrappers and since the mapping of Cathedral design data onto the Nelsis primitives for data organization such as viewtypes, design objects and streams was fairly straightforward, a design flow could easily be defined. The same holds for Chess [Lanneer95], a retargetable code generation environment, developed at the same institute. The design flow that has been developed for the Chess environment is shown in figure 6.27.

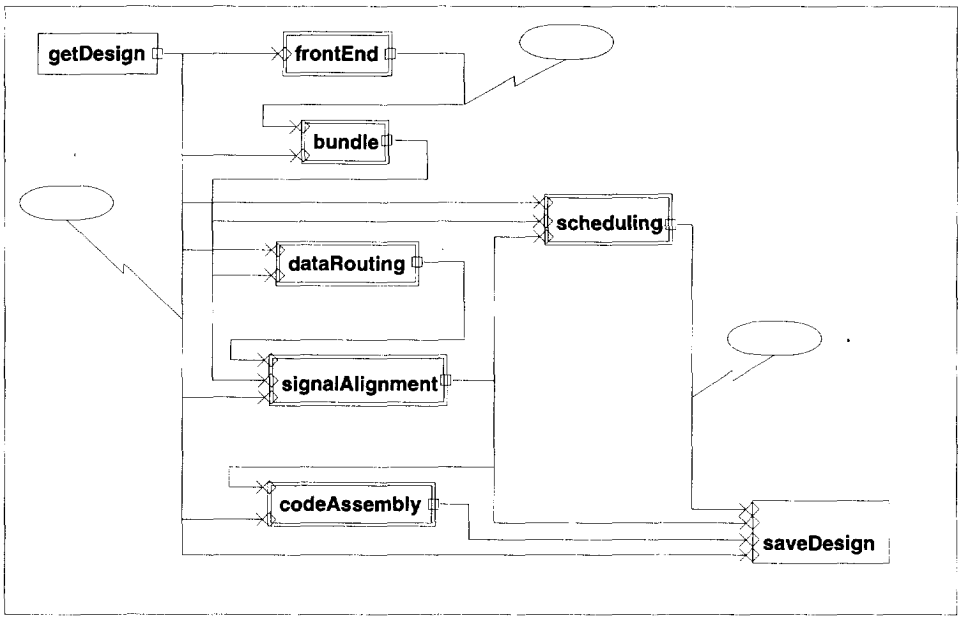


Figure 6.27. The design flow of the Chess environment.

The use of the Nelsis design flow system at Imec has been unique in the fact that both the encapsulation of the Cathedral-2nd and the Chess tools as well as the definition of the different design flows has been performed in a short time-frame and almost without the help of the Nelsis development team. This is an important observation with respect to the requirement to strive for a small setup time.

Another important observation is that even with a small number of design flow constructs a fairly complex design environment can be modeled. For instance, in the Chess design flow, each viewtype contains only one datatype and each activity creates one or more new design objects.

The use of the Chess design flow also demonstrates the power of combining versioning and design flow management. If, starting from a colored flow in which all design steps have been executed, one of the early steps in the design flow is reexecuted, the versioning system generates a new version of the output data of this design step. In addition, the flow coloring algorithm derives a new colored flow in which the new version appears and all downstream flow primitives are cleared. The new colored flow clearly shows that the new version has not been processed yet. However, the old version and all its derived data is still available and if it is selected, the flow coloring algorithm presents the old colored flow.

6.8.7 The Mimola Hardware Design System

At the University of Dortmund, Germany, a number of tools from the Mimola (*M*achine *I*ndependent *M*icr*O*programming *LA*nguage) hardware design system [Marwedel93] have been integrated into the Nelsis CAD framework. This environment comprises tools for interactive synthesis, microcode generation, testgeneration and simulation. The design flow that has been designed for this environment [Wente94] is shown in figure 6.28.

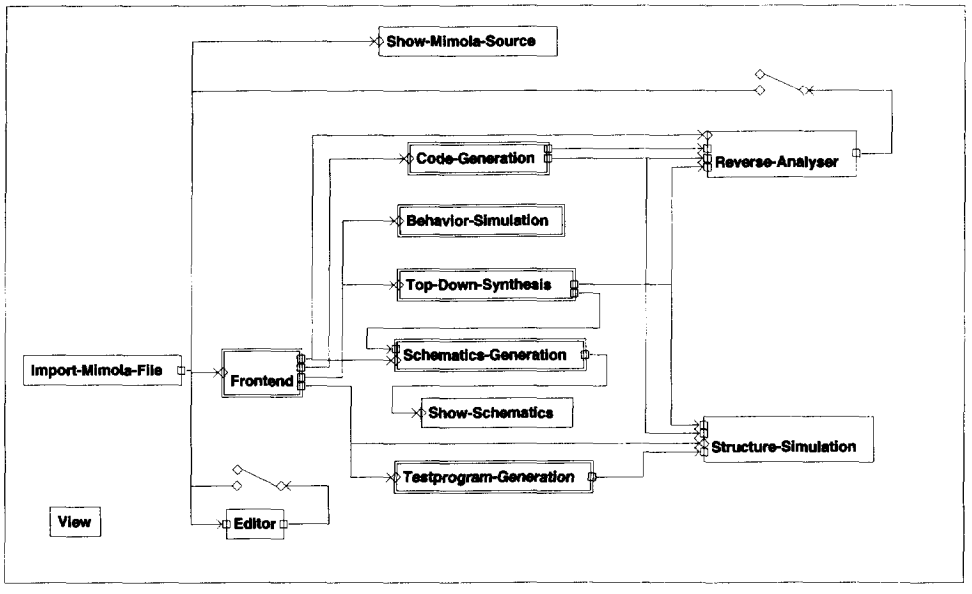


Figure 6.28. The design flow of the Mimola hardware design system.

The main reason for the integration of the Mimola system in Nelsis was the ability to express the design process in a design flow and to utilize the associated design flow control mechanisms. Due to the rather simple data organization in Mimola, the data management facilities of the Nelsis CAD framework were less important. The main conclusion of the integration project was that the ability to express and use a design flow was ideally suited for the Mimola design environment, which contains many different tools with implicit constraints on the order of usage. In addition, it appeared that the design flow configuration editor, FLUID, was a great help in designing, modifying and testing the design flow.

6.8.8 A Design Flow for System Simulation

At Philips, several design flows for different simulation purposes have been developed. One of these design flows, described in [Bingley94], combines a number of tools for the simulation of the design of a CD-player and its components and for inspecting the simulation results.

This project has been useful as it tested the Nelsis design flow system with a very large number of design objects. Although the design flow kernel had no problems processing and administering the large amount of run-time information, the design flow user interface appeared to have troubles when performing complex operations such as the browsing of a port or the derivation of a maximal colored flow. Due to the tremendous number of design objects related via the different flow graphs and the many different dependencies inside one particular compound flow graph, the response time for some of the browsing operations on the ports of this compound flow graph was too high. Especially the calculation of all design objects on one of the ports of the flow graph in question took too much time from a user interface point of view. However, if the same operation was applied in a colored flow where some design objects already had been assigned, the search space for the operation was reduced which speeded up the calculation of the browsing result. Besides the introduction of a number of low-level performance improvements in the design flow user interface, this experience also shows the need to develop efficient algorithms to implement the browsing operations presented in chapter 5 for compound flow graphs with complex internals. This issue has been described only briefly in chapter 5 and is still a topic for further research.

6.9 Conclusion

The benefits of using a design flow system become especially clear in complex design processes where designers perform their design tasks using a variety of different design tools. In such design systems the design tools are likely to be integrated into a CAD framework. Therefore, to maximally exploit the services of a design flow system in complex design processes, it is useful to make a design flow system an integral part of such a CAD framework. In this chapter we addressed the integration of the design flow system presented in the previous chapters into the Nelsis CAD framework.

To incorporate the design flow system into the Nelsis CAD framework, some of the general design flow concepts had to be refined. One refinement ensued from the requirement to perform fine-grain design flow management. For this, the

design flow system has to find out which activity runs a tool performs, while this tool is still running. We stated that it is better to recognize activity runs from data accesses than to force tool developers to extend their design tools with design flow-specific calls to specify what activities their tools perform. We feel that it is a major advantage to implement a design flow system in such a way that tools can simply read and write their design data in the usual way while the design flow system silently monitors and administers the design process in the background. Also, this is in line with our requirement for unobtrusiveness.

Another extension of the general design flow concepts relates to the organization of design data in Nelsis. To meet the extensive possibilities for organizing design data, the port construct in the general tool model has been split up into two types of ports. This enables the description of data access characteristics at the design object level as well as at the abstract datatype level. In addition, since we made the mapping of streamtypes to datatypes to viewtypes configurable, a design flow configurator can vary the preciseness for describing the data access characteristics of the tools from the coarse level of design objects (each viewtype consists of exactly one datatype) to the detailed level of streams (each datatype contains exactly one streamtype).

A third point of attention was to take care that the methods for flow-based user interaction ally to the general principles for user interaction in the Nelsis CAD framework. Since Nelsis users interact with the design system mainly in terms of design objects (for instance, design information is presented in terms of relationships between design objects and tools are invoked by referring to the design objects on which they should work), we defined the flow coloring and flow browsing operations in terms of design objects rather than at the level of datatypes.

The use of the design flow system in design environments for which the tool set changes during the design process, made us examine the effects of (re)definitions of the design flow configuration. Since in the Nelsis design flow model run-time information refers to elements of the design flow configuration, changes to the design flow configuration should work their way into the run-time information. However, if run-time information that is no longer consistent with the design flow configuration is simply invalidated, valuable design data may get lost. Therefore, we proposed a simple marking mechanism, that allows designers to see which design objects have been influenced by a change in the design environment without making these design objects worthless.

An important aspect of a design flow system is its integration into the architecture of a CAD framework. We described where the different design flow components should be placed in the component architecture of Nelsis, its distribution among different Unix processes and the use of a notification mechanism to update the colored flows displayed by the different design flow user interface processes in a distributed multi-user, multi-tasking design environment.

Besides the cooperation of the Nelsis data management features and the Nelsis user interface with the design flow system, it is also important that the design flow system works in harmony with other CAD framework services. We briefly discussed the interplay of design flow management with library management and with an access control facility. To support the definition of library projects from which design data can be imported into the project a designer is working in, we proposed to extend a design flow with a number of import activities. With respect to the integration of access control with design flow management we showed that a small extension of the information model enables the assignment of access permissions to flow graphs.

From the different design flows that have been designed for design systems in different application domains, we learned that in order to be widely applicable a design flow system should be highly configurable. One aspect that clearly shows the use of configurability is the use of the viewtype to datatype to streamtype mapping. In different application environments, different choices have been made for grouping streams into datatypes and datatypes into viewtypes. For instance, in the Mimola design flow each datatype consists of exactly one streamtype and a viewtype may contain many different datatypes. This means that in this application the mapping of streams into datatypes is not used at all. In the Chess design flow on the contrary, it was chosen to let each viewtype have only one datatype and to group up to 10 different streamtypes into one datatype.

A global observation that can be made from the many different design flows that we presented in the example section of this chapter is that the Nelsis design flow system is suited to model a range of different design environments. From the example design flows we conclude that a design flow system is particularly useful in design environments with many design tools with data and / or temporal dependencies and in design environments where design tools can operate in a multitude of operation modes. Especially in design environments with many different design tools that have a clearly defined execution order the use of a design flow system contributes to a better overview on the design process.

7

Conclusion

In this thesis we have developed a number of concepts for design flow management in CAD frameworks. These concepts can be used to build smarter design systems that help designers to better master the increasing complexity of today's design processes. We have described an extension of a CAD framework, called a *design flow system*, that exploits knowledge about the structure of the design process, described in a *design flow*, to help designers realize their design goals.

The major results of this thesis are a *design flow model* offering constructs to describe the structure of a design process in a design flow and a set of *design flow management functions* that assist designers in performing their complex design tasks.

With respect to the design flow model, we have searched for constructs that are *flexible, expressive* and *general* enough to describe the characteristics of any type of design process in detail, if these characteristics are known, or vaguely, if not. The design flow model has been used to describe the characteristics of a number of different design processes in different application areas with different design tools varying from simple to more complex. From this, we conclude that the constructs do satisfy these global characteristics.

With respect to the design flow management functions developed in this thesis, we specifically aimed at functionality in the areas of *design tracking, constraint enforcement, design assistance* and *design automation*. Detailed tracking of the design process is performed through the administration of tool runs, activity runs and data accesses and the visualization of this information in a *colored flow*. To make it possible to enforce design constraints we defined a number of *rules* that specify under which circumstances a tool is allowed to access data, with respect to the information configured in a design flow. The *flow coloring* paradigm, which describes how to represent part of the design state in the context of a design flow and which defines a number of flow coloring operations to browse through the

design state, is a valuable ground for designer assistance. Finally, the automation of (parts of) the design process is supported by the possibility to define and execute *design schedules*.

In the introduction of this thesis we stated that we aimed at a design flow system that supports a *wide range of tools*, that performs *fine-grain design flow management*, that operates in an *unobtrusive* way and that has a *small setup time*. We can now more specifically formulate why the design flow system described satisfies these global requirements:

- support for a *wide range of tools*:
  The design flow model presented contains a number of constructs to describe tools in a flexible way. For instance, the activity concept, the multiplicity construct and the possibility to define generic datatypes in a datatype hierarchy make it possible to describe tools varying from interactive to batch and from specific to generic.

- *fine-grain design flow management*:
  Rather than assisting designers at the coarse-grain level of tool runs, the design flow system operates at a fine-grain level so that each unit of tool operation meaningful to the end-user, called a *design function*, can be tracked, checked and automated individually.

- *unobtrusiveness*:
  The design flow system is unobtrusive in two different ways. First, due to the proper integration of the design flow system into the architecture of a CAD framework based design system, designers are not forced to perform their design activities via the design flow user interface, while the design flow system still correctly tracks the design process and enforces the design constraints. Second, design tools do not need to be extended with special calls to the design flow system in order to take advantage of the design flow management functionality.

- a *small setup time*:
  Using the *tracing* or *learning* mode as described in chapter 2, the task to configure the design flow system for a simple design process may be easy. However, for complex design processes the interference of a design flow configurator, who manually defines a design flow, is still indispensable. To facilitate manual configuration the design flow system offers a *design flow editor*.

The concepts for design flow management have been implemented in the Nelsis CAD framework and the resulting system has been used in a number of different design applications. For each of the applications a design flow has been defined, which describes the structure of the design process in terms of the constructs offered by the Nelsis design flow model. These design flows accommodate the knowledge of the different design applications that the design flow system needs to assist designers in executing the design process.

In the introduction of this thesis we stated that we expect design flow management to make design systems smarter. We used the word smarter to indicate that a design system with design flow management, using its knowledge about the design process, can offer functionality that cannot be offered in systems that do not have such knowledge. Although it is actually a more philosophical question whether we may term a specific system smart or not, the design flow system presented does meet our expectations. Based on the knowledge recorded in a design flow it offers functionality such as:

- Knowing the characteristics of the design tools, the design flow system can track the design process in terms of activities and data accesses and present (parts of) the tracked design state in a colored flow.
- Knowing the structure of the design process, the design flow system can answer complicated questions about the design state, issued by designers via (colored) flow-based user interaction.
- Knowing the constraints that hold in a certain design process, the design flow system can keep designers from making certain design errors.
- Knowing the execution conditions of tools, the design flow system can inform designers of the applicability of the design functions of tools.
- Knowing the inherent relationships between tools and data, the design flow user interface can select and present those parts of the design state a designer is most interested in.
- Knowing the control characteristics of the tools and the meaning of these characteristics in relation to the design process, the design flow system can help designers to invoke tools at the right moment and in the right way.
- Knowing the dependencies between the different activities of tools and the way design tools can be controlled, a design flow system can automatically bring a design into a required state.

- Knowing the relationships between low-level design functions of tools and high-level design phases (as expressed by a hierarchical design flow), the design flow system can help designers to operate at a more abstract level.

From this list it is clear that the key to the smartness ascribed to our design flow system is the knowledge in a design flow. To make design systems even smarter, in our view, their knowledge about the design process should be further extended and functions should be defined to exploit this knowledge for user assistance. However, this is not the only aspect of smartness. Another important aspect is the way the system collects its knowledge. Currently, we expect most of this knowledge to be brought in by a design flow configurator. The use of the Nelsis design flow system in a number of different application domains has shown that this task may be rather tricky. Not only does a design flow configurator need to have detailed knowledge of the operation of the tools and the characteristics of the design process as a whole, he also needs to be familiar with the constructs and concepts offered by the design flow system. Therefore, it is desirable to make the configuration task more easy or even unnecessary. It would certainly contribute to the smartness of design systems if they would have some kind of learning facility which enables them to gather and maintain this type of knowledge themselves. The learning mode described in section 2.4.3 is only a first step in this direction. Actually, we feel that this topic should be at the top of our list for further research.

Another topic that deserves further attention is how to cope with changing design environments. The marking strategy mentioned in chapter 6 offers only one of the solutions to handle this problem. It might be useful to provide means to automatically update the administered state of design upon changes in the design flow.

The observation that some people actually first sketch a design flow before they start integrating or encapsulating their design tools into a CAD framework, made us come up with the idea that the learn mode described in this thesis could also be used in the other direction: to generate tool encapsulation or integration code from a specified design flow.

Finally, the concepts presented in this thesis have been developed with the electronic design process in mind. However, since it is our experience that the design flow model is general enough to describe a wide range of different CAD tools, it might be useful to investigate whether these concepts apply to other application areas as well.

A

Appendix

This appendix contains a brief and rather informal introduction to the semantic data modeling technique OTO-D. It is meant to provide readers of this thesis who are not familiar with OTO-D with some basic knowledge of this data modeling technique. Since we use OTO-D in this thesis mainly as a tool for defining the information structure of our system, we limit the discussion to those constructs that actually have been used and let the more fundamental considerations for what they are. For a more extensive overview of OTO-D we refer to [Bekke92].

The abbreviation OTO-D stands for *O*bject *T*ype *O*riented *D*ata modeling. This name refers to one of its basic principles: that, to describe the dynamic aspects of the reality, data modeling must be based on a *type* concept rather than on a *set* concept as in most classical approaches. Where a set consists of a collection of definite, distinct objects, a type defines a number of properties of a dynamic collection of objects.

Another important principle in OTO-D is the use of formally defined diagrams to present complicated models in a sound and understandable way. In this thesis we make extensive use of these diagrams.

The constructs offered by OTO-D can be roughly divided into constructs for *data definition* and constructs for *data manipulation*. The data definition language (DDL) can be used to describe the structure and properties of a specific application environment in an *information model* (also known as a conceptual model or data schema). The purpose of the *data manipulation language* (DML) is to add, update, delete and retrieve data to, in or from a database organized according to a specific information model. Below, we describe the OTO-D data definition and data manipulation constructs in more detail.

Data Definition

Data definition is the derivation of an information model that represents a portion of the real world and that consists of a description of relevant objects. Central in deriving such an information model is the notion of *abstraction*. The OTO-D modeling technique supports three types of abstraction.

The first type of abstraction is called *classification*. Classification is the determination of the relevant properties of relevant objects. For instance, suppose that we derive an information model for the administration of a field hockey club, where one would like to store information about the players. The first thing to do is to determine the relevant properties that characterize the players, such as *Name*, *Address*, *Position* in the field and the *Team* a player is a member of.

The second type of abstraction supported by OTO-D is *aggregation*. Aggregation is the type of abstraction where a certain number of different properties are combined into a *type* (also called object type), which can be discussed without referring to its properties. For instance, we can talk about individual players without mentioning their addresses. These individual players are *instances* of the type *Player*. A type is defined by listing its properties. Assuming that each player in the above example is a member of exactly one team and always plays at the same position, this leads to the following type definition:

TYPE Player = Name, Address, Position, Team

Graphically, properties and types are represented by rectangles with their name in it and aggregation is represented by a line connecting the bottom center to the top center of the rectangles involved. For the above type definition this leads to the picture as shown in figure A.1.

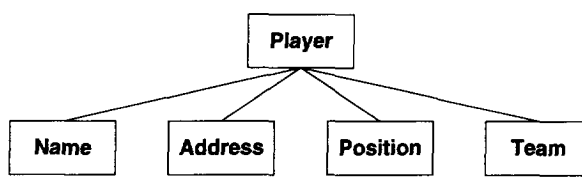


Figure A.1. A player is characterized by its name, address, position and team.

In OTO-D, a type such as the *Player* type in the above example, can in itself be regarded as a new property of another type. Such a collection of type definitions is called an *abstraction hierarchy*. As an example, a *Team* in itself may be a type characterized by its *TeamName* and the *League* in which it is playing:

TYPE Team = TeamName, League
TYPE Player = Name, Address, Position, Team

This leads to the graphical representation as shown in figure A.2.

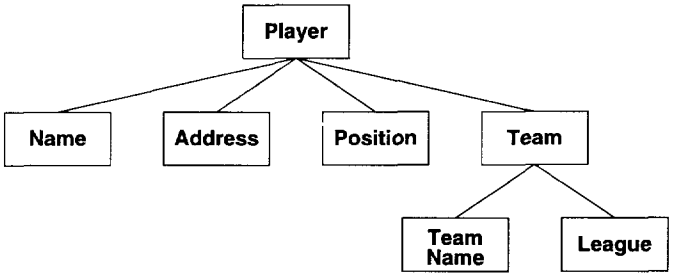


Figure A.2. An abstraction hierarchy with 2 aggregations.

The third type of abstraction supported by OTO-D is *generalization*. Generalization can be used to describe different types with similar properties. For example, we might want to describe persons that are not players but that do have a name and an address. In that case, the type *Person* is a generalization of the type *Player*, or expressed in the opposite direction, a player is a special kind of person (this is called *specialization*). Specialization is described in a type definition by surrounding the property involved with square brackets. For our example this leads to the following type definitions:

TYPE Team = TeamName, League
TYPE Person = Name, Address
TYPE Player = [Person], Position, Team

The graphical representation of generalization is a line connecting the corners of the object types involved. The above abstraction hierarchy is represented in figure A.3.

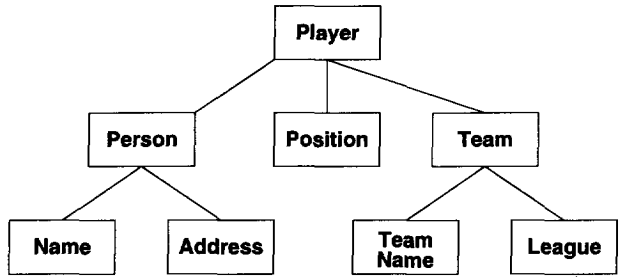


Figure A.3. An abstraction hierarchy with aggregation and generalization.

To distinguish between properties of a type with the same name it may be necessary to assign a name to the relationship between a type and its property. In OTO-D, such a name is known as a *role*. For instance, if we like to administer the matches between teams and their resulting scores, we have to use a role to distinguish the receiving team from the visiting team. This is described as follows:

TYPE Team = TeamName, League
TYPE Person = Name, Address
TYPE Player = [Person], Position, Team
TYPE Match = Receiving-Team, Visiting-Team, Score

In the graphical representation (the first letter of) a role is shown at the line indicating the relationship involved. This is shown in figure A.4.

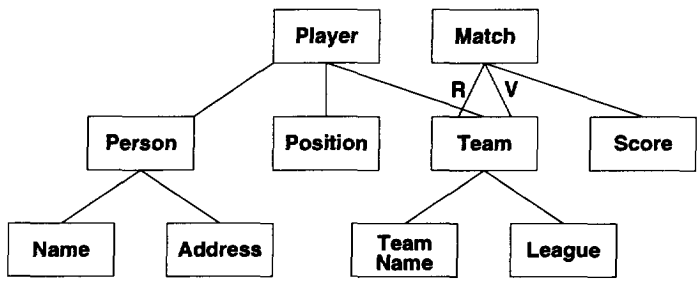


Figure A.4. Roles are used to distinguish between properties with the same name.

In OTO-D, type definitions are supposed to satisfy two important integrity rules.

- *relatability*:
  Each property in a type definition is related to one and only one equally named type, while every type may correspond to various properties.

  At the instance level relatability implies that an attribute value is related to one and only one instance in the related type.

- *convertibility*:
  Each type definition is unique, i.e. there are no type definitions carrying the same name or the same set of properties.

OTO-D offers two important constructs to specify the inheritance of properties in an abstraction hierarchy:

- The ITS construct describes property inheritance in the downward direction. It can be used to refer to properties of types via multiple aggregation or generalization relationships. For instance, the phrase

  ... Player ITS Person ITS Name ...

  refers to the property *Name* of a *Player* inherited via the type *Person*.

- The PER construct describes property inheritance in the upward direction. Since the aggregation and generalization relationships followed in the upward direction may lead to a set of instances, this construct must be used in combination with a set function such as max, min, total, count, some, any or nil. For instance, the phrase

  ... COUNT Player PER Team ...

  refers to the number of players of a team.

The use of the ITS and PER constructs will become clear in the examples below.

Except for the definition of the types in terms of their properties, the definition of an information model may also take the specification of a number of *constraints*. In this thesis we are only interested in one type of constraints: *static constraints*. A static constraint is a statement that is true for every valid database state. We introduce the OTO-D constructs to be used for the specification of static constraints by example. For example, a useful static constraint could be to demand that a *Match* can only be played between two *Teams* of the same *League*. In OTO-D this is described using the ASSERT construct in the following way:

```
ASSERT Match ITS ValidMatch (TRUE) =
  Receiving-Team ITS League = Visiting-Team ITS League
```

Another useful static constraint could be to demand that a team cannot play against itself:

```
ASSERT Match ITS ValidMatch (TRUE) =
  Receiving-Team != Visiting-Team
```

or to demand that the number of players of a team is 11 or more:

```
ASSERT Team ITS ValidTeam (11..*) =
  COUNT Player
  PER Team
```

OTO-D does not offer any graphical equivalents for the ASSERT construct.

Data Manipulation

The data manipulation language offers constructs to add, update, delete and retrieve data to, in or from a database that is organized according to a specific information model, or to extend the database with some derived attribute. An example of the first type of construct is:

```
GET Player ITS Person ITS Name
  WHERE Position = 'goalkeeper'
```

This DML command returns all registered keepers.

An example of the second type of construct is the command

```
EXTEND Team WITH NumberOfPlayers
  COUNT Player
  PER Team
```

This causes the type *Team* to be extended with a property that describes the number of players per team.

OTO-D does not offer any graphical equivalents for the GET and the EXTEND constructs.

References

Allen90. W. Allen, D. Rosenthal, and K. Fiduk, "Distributed Methodology Management for Design-in-the-Large", *Proc. 8th IEEE/ACM International Conference on CAD*, pp. 346-349 (1990).

Allen91. W. Allen, D. Rosenthal, and K. Fiduk, "The MCC CAD Framework Methodology Management System", *Proc. IEEE 28th Design Automation Conference*, pp. 694-698 (1991).

Baltes94. H. Baltes, "JCF 4.0: Detailed Functional Specification", SP2 Milestone: M2.DPM.5R, ESPRIT Project 7364, Siemens (March 1994).

Barnes92. T.J. Barnes, D. Harrison, A.R. Newton, and R.L. Spickelmier, *Electronic CAD Frameworks,* Kluwer Academic Publishers (1992).

Bartels94. G. Bartels, P. Kist, K. Schot, and M. Sim, "Flow Manager Requirements of a Test Harness for Testing the Reliability of an Electronic CAD System", *Proc. European Design and Test Conference*, pp. 605-609 (Feb / March 1994).

Beggs92. R. Beggs, "Automated Design Decision Support System", *Proc. 29th ACM/IEEE Design Automation Conference*, pp. 506-511 (June 1992).

Bekke92. J.H. ter Bekke, *Semantic Data Modeling,* Prentice Hall, Englewood Cliffs, N.J. (1992). ISBN 0-13-806050-9.

Bingley90. P. Bingley and P. van der Wolf, "A Design Platform for the NELSIS CAD Framework", *Proc. 27th ACM/IEEE Design Automation Conference*, pp. 146-149 (June 1990).

Bingley92. P. Bingley, K.O. ten Bosch, and P. van der Wolf, "Incorporating Design Flow Management in a Framework Based CAD System", *Proc. IEEE/ACM International Conference on CAD - 92*, pp. 538-545 (Nov 1992).

Bingley94. P. Bingley and W. van der Linden, "Application of Framework Technology in System Simulation Environments", *Proc. Seminar Database Systems and Applications for the 90's*, (October 1994).

Bosch91. K.O. ten Bosch, P. Bingley, and P. van der Wolf, "Design Flow Management in the NELSIS CAD Framework", *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 711-716 (June 1991).

Bosch93. K.O. ten Bosch, P. van der Wolf, and P. Bingley, "A Flow-Based User Interface for Efficient Execution of the Design Cycle", *Proc. IEEE/ACM International Conference on CAD - 93*, pp. 356-363 (Nov 1993).

Bosch94. K.O. ten Bosch, P. van der Wolf, and A. van der Hoeven, "Design Flow Management: More than Convenient Tool Invocation", *Proc. 4th Int. IFIP WG 10.5 Working Conference on Electronic Design Automation Frameworks*, pp. 137-146 (preprint version) (Nov 1994).

Bretschnei90. F. Bretschneider, C. Kopf, H. Lagger, A. Hsu, and E. Wei, "Knowledge Based Design Flow Management", *Proc. IEEE/ACM International Conference on CAD - 90*, pp. 350-353 (Nov 1990).

Bretschnei92. F. Bretschneider, "A Process Model for Design Flow Management and Planning", *PhD thesis*, (Juli 1992).

Brockman91. J.B. Brockman and S.W. Director, "The Hercules CAD Task Management System", *Proc. 9th IEEE/ACM International Conference on CAD*, pp. 254-257 (1991).

Brockman92. J.B. Brockman, T.F. Cobourn, M.F. Jacome, and S.W. Director, "The Odyssey CAD Framework", *IEEE DATC Newsletter on Design Automation*, (Spring 1992).

Cadence90. Cadence Design Systems, "Framework Technology for the 1990s", *Cadence Marketing Materials*, (1990).

Câmara94. J. Camara and H. Sarmento, "AutoCap: An Automatic Tool Encapsulator", *Proc. 4th Int. IFIP WG 10.5 Working Conference on Electronic Design Automation Frameworks*, pp. 31-40 (preprint version) (Nov 1994).

Casotto90. A. Casotto, A.R. Newton, and A. Sangiovanni-Vincentelli, "Design Management based on Design Traces", *Proc. 27th ACM/IEEE Design Automation Conference*, pp. 136-141 (1990).

CDE93. Common Operating System Environment (COSE), *Common Desktop Environment, Development Reference.* 10/93 Snapshot, Public distribution.

CFIfar93. CFI Architecture Working Group, "Framework Architecture Reference, Version 1.2", *CAD Framework Initiative*, (June 1993).

CFIitc92. CFI Inter-Tool Communication Working Group, "Inter-Tool Communication Programming Interface, Version 1.0.0", *CAD Framework Initiative*, (1992).

CFItes92. CAD Framework Initiative, Inc. Design Methodology Management Technical Subcommittee, *Tool Encapsulation Specification, Version 1.0.0.* 1992.

CFItesdisc93. CAD Framework Initiative, *Discussions CFI TES 1.1 Working Group*, (email distribution list: cfi-tes@cfi.org) June 1993 - Feb 1994.

CFIugo90. CFI Architecture Technical Subcommittee, "CAD Framework Users, Goals, and Objectives, Version 0.92", *CAD Framework Initiative*, (Dec 1990).

Chiueh93. T. Chiueh and R.H. Katz, "A Distributed Transaction Facility for Design Task Management", *Proc. European Design Automation Conference*, pp. 46-50 (1993).

Dewilde86. P. Dewilde, ed., *The Integrated Circuit Design Book: Papers on VLSI Design Methodology from the ICD-NELSIS Project,* Delft University Press, Delft, The Netherlands (1986).

Disselkoen95. P.L. Disselkoen, "Incorporating Flow Management into ANT", Master's Thesis, Delft University of Technology, Delft (Aug 1995).

EDIF87. EDIF, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual", *EDIF Steering Committee*, Electronic Industries Association, (1987).

Fiduk90. K.W. Fiduk, S. Kleinfeldt, M. Kosarchyn, and E.B. Perez, "Design Methodology Management - A CAD Framework Initiative Perspective -", *Proc. IEEE 27th Design Automation Conference*, pp. 278-283 (1990).

Gedye88. D. Gedye and R.H. Katz, "Browsing the Chip Design Database", *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 269-274 (June 1988).

Hamer90. P. van den Hamer and M.A. Treffers, "A Data Flow Based Architecture for CAD Frameworks", *Proc. 8th IEEE/ACM International Conference on CAD*, pp. 482-485 (1990).

Hamer91. P. van den Hamer, K.O. ten Bosch, P. Bingley, M.A. Treffers, and P. van der Wolf, "A Comparison of Two Approaches to Design Flow Management by Data Schema Analysis", JCF Project Deliverable, SP1, Philips Research Laboratories & Delft University of Technology (June 1991).

Harrison86. D.S. Harrison, P. Moore, R.L. Spickelmier, and A.R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment", *Proc. IEEE ICCAD-86*, pp. 24-27 (1986).

Held93. P.C. Held, P. Dewilde, E.F. Deprettere, and P. Wielage, "HiFi: From Parallel Algorithm to Fixed-Size VLSI Processor Array", pp. 71-94 in *Application-Driven Architecture Synthesis*, ed. F. Catthoor and L. Svensson, Kluwer Academic Publishers, Boston (1993).

Held94. P.C. Held, "Configuration of the NELSIS CAD Framework for the HiFi design system", Technical Report ET/NT/HIFI-15, Dept. Electrical Engineering, Delft University of Technology, Delft, The Netherlands (1994).

Hoeven94. A. van der Hoeven, K.O. ten Bosch, P. van der Wolf, and R. van Leuken, "A Flexible Access Control Mechanism for CAD Frameworks", *Proc. European Design Automation Conference*, pp. 188-193 (Sep 1994).

Hübel92. C. Hübel, D. Ruland, and E. Siepmann, "On modeling Integrated Design environments", *Proc. Euro-Dac*, pp. 452-458 (1992).

Jackson83. M. Jackson, *System Development*, Prentice Hall, Englewood Cliffs, N.J. (1983).

Jacome92. M.F. Jacome and S.W. Director, "Design Process Management for CAD Frameworks", *Proc. 29th ACM/IEEE Design Automation Conference*, pp. 500-505 (June 1992).

Janni86. A. Di Janni, "A Monitor for Complex CAD Systems", *Proc. 23rd ACM/IEEE Design Automation Conference*, pp. 145-151 (1986).

JCFarch91. JCF Task Force Architecture, "Jessi-Common-Framework Architecture Specification", *Jessi-Common-Frame project (Esprit 5082 / 7364)*, (June 1991).

JCFgloss92. JCF SP2 Task Forces, "Jessi-Common-Framework Glossary SP2", *Jessi-Common-Frame project (Esprit 5082 / 7364)*, (Feb 1992).

Kleinfeldt94. S. Kleinfeldt, M. Guiney, J.K. Miller, and M. Barnes, "Design Methodology Management", *Proc. of the IEEE* **82**(2) pp. 231-250 (Feb 1994).

Kupitz92. E. Kupitz and J. Tacken, "DECOR - Tightly Integrated Design Control and Observation", *Proc. IEEE/ACM International Conference on CAD - 92*, pp. 532-537 (Nov 1992).

KweeChrist95. E. Kwee-Christoph, F. Feldbusch, R. Kumar, and A. Kunzmann, "Generic Design Flows for Project Management in a Framework Environment", *Proc. European Design and Test Conference*, pp. 280-284 (1995).

Lanneer90. D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man, "Open-ended System for High-Level Synthesis of Flexible Signal Processors", *Proc. European Design Automation Conference*, pp. 272-276 (March 1990).

Lanneer95. D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: Retargetable Code Generation for Embedded DSP Processors", pp. 85-101 in *Code Generation for Embedded Processors*, ed. P. Marwedel and G. Goossens, Kluwer Academic Publishers (1995).

Lepoeter93. K. Lepoeter and M. Schlueper, "Feasibility Study: The use of NELSIS within IGS", Report RWB-558-KL-93023, Philips Electronic Design & Tools (Feb 1993).

Liebisch92. D. C. Liebisch and A. Jain, "JESSI COMMON FRAMEWORK Design Management -- The Means to Configuration and Execution of the Design Pocess", *Proc. Euro-Dac*, pp. 552-557 (1992).

Lopez92. J.C. Lopez, M.F. Jacome, and S.W. Director, "Design Assistance for CAD Frameworks", *Proc. Euro-Dac*, pp. 494-499 (1992).

Marwedel93. P. Marwedel and W. Schenk, "Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MSS", *Proc. European Design Automation Conference*, pp. 63-69 (Feb 1993).

Mathys93. Y. Mathys and V. Vasudevan, "Tracking design methodology in DAMOCLES", *Proc. European Design Automation Conference*, pp. 51-56 (1993).

Meijs87. N. van der Meijs, T.G.R. van Leuken, P. van der Wolf, I. Widya, and P. Dewilde, "A Data Management Interface to Facilitate CAD/IC Software Exchanges", *Proc. IEEE International Conference on Computer Design '87*, pp. 403-406 (1987).

Nelsis93. Nelsis Documentation, "Fluid, a graphical editor for Nelsis design flow configurations", User's Manual, Delft University of Technology (1993).

Post87. F.H. Post, "Cad / Cam I", Lecture Notes Cad / Cam (in Dutch), Delft University of Technology, Delft (January 1987).

ProFrame94. IBM, "Electronic Design Automation Solutions; ProFrame", *IBM Marketing Materials*, (1994).

Rumsey92. M. Rumsey and C. Farquhar, "Unifying Tool, Data and Process Flow Management", *Proc. EURO-DAC 92*, pp. 500-505 (Sept 1992).

Sarmento91. H. Sarmento and P. R. dos Santos, "PACE - A Framework for Electronic Design Automation", *Proc. IFIP WG 10.2 Workshop on Electronic Design Automation Frameworks, Nov 1990*, pp. 85-97 North-Holland, (1991).

Schot92. C.A. Schot, M.N. Sim, and P.M. Kist, "ANT - A Test Harness for the NELSIS CAD System", *Proc. European Design Automation Conference*, pp. 506-511 (Sep 1992).

Sommervill85. I. Sommerville, *Software Engineering*, Addison-Wesley Publishing Company (1985).

Sutton93. P.R. Sutton, J.B. Brockman, and S.W. Director, "Design Management Using Dynamically Defined flows", *Proc. IEEE 30th Design Automation Conference*, (1993).

Tanaka90. T. Miyazaki (Tanaka), T. Hoshino, and M. Endo, "A CAD Process Scheduling Technique", *Proc. IEEE/ACM International Conference on CAD - 90*, pp. 354-357 (Nov 1990).

ToolTalk91a. Ricki Frankel, "Introduction to the ToolTalk Service, a white paper", *Sun Microsystems, Inc*, (1991).

ToolTalk91b. Ricki Frankel, "ToolTalk in Electronic Design Automation, a white paper", *Sun Microsystems, Inc*, (1991).

Vasudevan92. V. Vasudevan, Y. Mathys, and J. Tolar, "Damocles: An Observer-Based Approach to Design Tracking", *Proc. IEEE/ACM International Conference on CAD - 92*, pp. 546-551 (Nov 1992).

Wagner92. F.R. Wagner and others, "Design Version Management in the STAR framework", *Proc. 3th Int. IFIP WG 10.5 Working Conference on Electronic Design Automation Frameworks*, pp. 85-97 (1992).

Wente94. D. Wente, "Integration von Hardware-Entwurfswerkzeugen in ein CAD-Framework", Master's Thesis, University of Dortmund, Dortmund, Germany (Jan 1994).

Widya88. I. Widya, T.G.R. van Leuken, and P. van der Wolf, "Concurrency Control in a VLSI Design Database", *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 357-362 (June 1988).

Wolf88. P. van der Wolf and T.G.R. van Leuken, "Object Type Oriented Data Modeling for VLSI Data Management", *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 351-356 (June 1988).

Wolf94a. P. van der Wolf, *CAD Frameworks: Principles and Architecture,* Kluwer Academic Publishers, Boston/Dordrecht/London (1994). ISBN: 0-7923-9501-8

Wolf94b. P. van der Wolf, K.O. ten Bosch, and A. van der Hoeven, "An Enhanced Flow Model for Constraint Handling in Hierarchical Multi-View Design Environments", *Proc. IEEE/ACM International Conference on CAD - 94*, pp. 500-507 (Nov 1994).

Samenvatting

Ontwerp Stroom Beheersing in CAD Frameworks

Het ontwerpen van complexe elektronische systemen gebeurt tegenwoordig veelal door teams van ontwerpers die hiervoor een scala van geavanceerde software programma's (tools) tot hun beschikking hebben. Om de grote hoeveelheden data die hiermee gemoeid zijn goed te beheren, het grote aantal steeds wisselende tools maximaal te benutten en de samenwerking tussen de individuele ontwerpers goed te laten verlopen, is het noodzakelijk om behalve de ontwerphandelingen zelf ook het gehele *ontwerpproces* softwarematig te ondersteunen. Dit heeft geleid tot het ontstaan van zogenaamde *CAD frameworks*, software systemen die een algemene werkomgeving bieden met faciliteiten zoals versie beheer, hiërarchisch ontwerp, het controleren van concurrente toegang tot data en het beheren van verschillende representaties van een ontwerp.

In dit proefschrift beschrijven we een andere veelbelovende CAD framework faciliteit, genaamd *design flow management*, wat zich laat vertalen als het beheersen van het ontwerpproces met behulp van *design flows* (ontwerp stroom diagrammen). Het idee achter design flow management is dat de karakteristieken van een ontwerpproces worden beschreven in een design flow, welke als basis dient voor een verdere ondersteuning van de ontwerpactiviteiten. Deze ondersteuning bestaat onder meer uit het volgen en representeren van het ontwerpproces in de context van een design flow, het eventueel afdwingen van de in de design flow vastgelegde beperkingen, het assisteren van de ontwerpers door hen de mogelijke en onmogelijke vervolgstappen te laten zien en het automatisch executeren van delen van het ontwerpproces.

In dit proefschrift ontwikkelen we een design flow (management) systeem dat geschikt is voor een groot aantal verschillende soorten tools en dat de ontwerpers op een voldoende gedetailleerd niveau assisteert. Een andere eigenschap van het design flow systeem is dat ontwerpers noch tool ontwikkelaars gedwongen worden hun werkwijze aan te passen, terwijl ze toch maximaal kunnen profiteren van de

voordelen van design flow management. Een laatste eigenschap tenslotte is dat de tijd benodigd om het systeem te configureren zo klein mogelijk is gehouden.

Een groot deel van dit proefschrift is gewijd aan het ontwikkelen van een bruikbaar *design flow model*, dat een aantal constructies biedt om ontwerpprocessen te beschrijven. We streven naar een model dat *algemeen* genoeg is om de eigenschappen van zeer verschillende ontwerpprocessen te kunnen beschrijven, *flexibel* genoeg om ook tools voor welke de karakteristieken vooraf niet kunnen worden bepaald te kunnen beschrijven en *expressief* genoeg om interessante eigenschappen van het ontwerpproces in detail te kunnen beschrijven. Omdat tools in verschillende ontwerpprocessen kunnen worden gebruikt is het zinvol de constructies voor het beschrijven van de eigenschappen van de individuele ontwerp tools te scheiden van constructies die het ontwerpproces als geheel beschrijven. Het geheel van constructies van de eerste soort noemen we het *tool model*.

Voor wat betreft het tool model zijn er twee soorten eigenschappen die in het bijzonder dienen te worden beschreven: de data access karakteristieken van de tools en hoe tools aangeroepen en gecontroleerd dienen te worden. Omdat veel tools gedurende één tool executie meerdere ontwerpfuncties kunnen uitvoeren en omdat tools veelal op meerdere verschillende manieren gebruikt kunnen worden, introduceren we het begrip *activiteit* als een abstractie van een ontwerp functie van een tool en bieden we constructies om de eigenschappen van tools per activiteit te beschrijven.

Waar het tool model is gericht op de beschrijving van de eigenschappen van afzonderlijke tools, richt het design flow model zich meer op de beschrijving van de structuur van een ontwerpproces als geheel. Dit omvat onder meer de afhankelijkheden tussen de verschillende activiteiten van de tools, een hiërarchische organisatie van design flows en informatie over de relaties tussen de data betrokken bij het ontwerp.

Het succes van een design flow systeem wordt mede bepaald door de manier van gebruikers interactie, gehanteerd in de *design flow user interface*. Een belangrijk aspect hiervan is de representatie op basis waarvan interactie plaatsvindt. We komen tot de conclusie dat een zogenaamde *colored flow* (gekleurd stroom diagram) een bruikbare representatie is. Op basis van deze colored flows definiëren we een aantal nuttige operaties rechtstreeks beschikbaar aan de ontwerpers, zoals het inspecteren van (delen van) de ontwerpstatus, het automatisch afleiden van een colored flow die juist die delen van de ontwerpstatus laat zien waarin de ontwerper het meest geïnteresseerd is en het definiëren en executeren van zogenaamde *design schedules* (ontwerp schema's).

Naast de theoretische onderbouwing van het design flow systeem beschrijven we in dit proefschrift tevens een implementatie van dit systeem in het reeds bestaande Nelsis CAD framework, ontwikkeld aan de technische universiteit Delft. Om naadloos met het Nelsis CAD framework te kunnen samenwerken, moeten sommige van de algemene design flow concepten verfijnd worden. Dat dit alles ook daadwerkelijk een bruikbaar systeem oplevert, illustreren we aan de hand van een aantal design flows zoals gedefinieerd in verschillende ontwerpsystemen gebruik makend van het Nelsis design flow systeem.

Olav 'ten Bosch

Acknowledgements

This research was supported in part by the commission of the EC under project 7364 (JESSI-Common-Frame). I want to thank all who supported me when writing this thesis, especially Pieter van der Wolf for his many useful comments, professor Dewilde for being my promotor, all other members of the committee for their comments, Corrie Boers and Ank Russo for the secretarial support, all people from DDTC who participated in the development of the Nelsis CAD framework: Peter Bingley, Edwin Essenius, Simon de Graaf, Alfred van der Hoeven, Peter Kist, Rene van Leuken, Peter van Putte, Kees Schot, Mattie Sim, Wim Tiwon, Peter van der Wekken and Pieter van der Wolf, the people from Philips for evaluating and discussing our prototype design flow systems: Kees Lepoeter, Peter van den Hamer, Menno Treffers and Peter Bingley, all other people who have used our software and provided us with useful input, Onno Zwaagstra for the many common-sense discussions about real life applications of (my) academic ideas, and, of course, Ingrid.

Curriculum Vitae

Olav ten Bosch was born in 1965 in Leiden, the Netherlands. After graduating at the "Stedelijk Gymnasium" in his birthplace (1984), he started his scientific career as a student at the department of Technical Mathematics and Informatics of the Delft University of Technology. As part of his study he worked for several months (1988) at the institute for testing electrical materials (KEMA) located in Arnhem, the Netherlands, where he contributed to the development of an information system for analyzing the occurrences of lightnings in the Netherlands. Back in Delft, the author chose to finish his study at the theoretical computer science group headed by professor van Westrhenen. His master's thesis was about non-monotonic reasoning using the so-called preference logic. Although the results of this work have been implemented in a so-called assumption-based truth maintenance system (ATMS), the lack of clearly defined application areas made the author turn his interest into other directions. He became involved in research in (electronic) CAD frameworks, carried out at the network theory section of the department of electrical engineering in corporation with the DIMES research institute. This research has been supported by the European JESSI project. In 1990 the author started to work on the issue of design flow management, which finally resulted in the thesis that you are reading at this very moment. Although we cannot look ahead too much, the future interests of the author will be in the CAD framework or process management area. However, the originally chosen direction of automated reasoning and artificial intelligence also keeps attracting his attention.