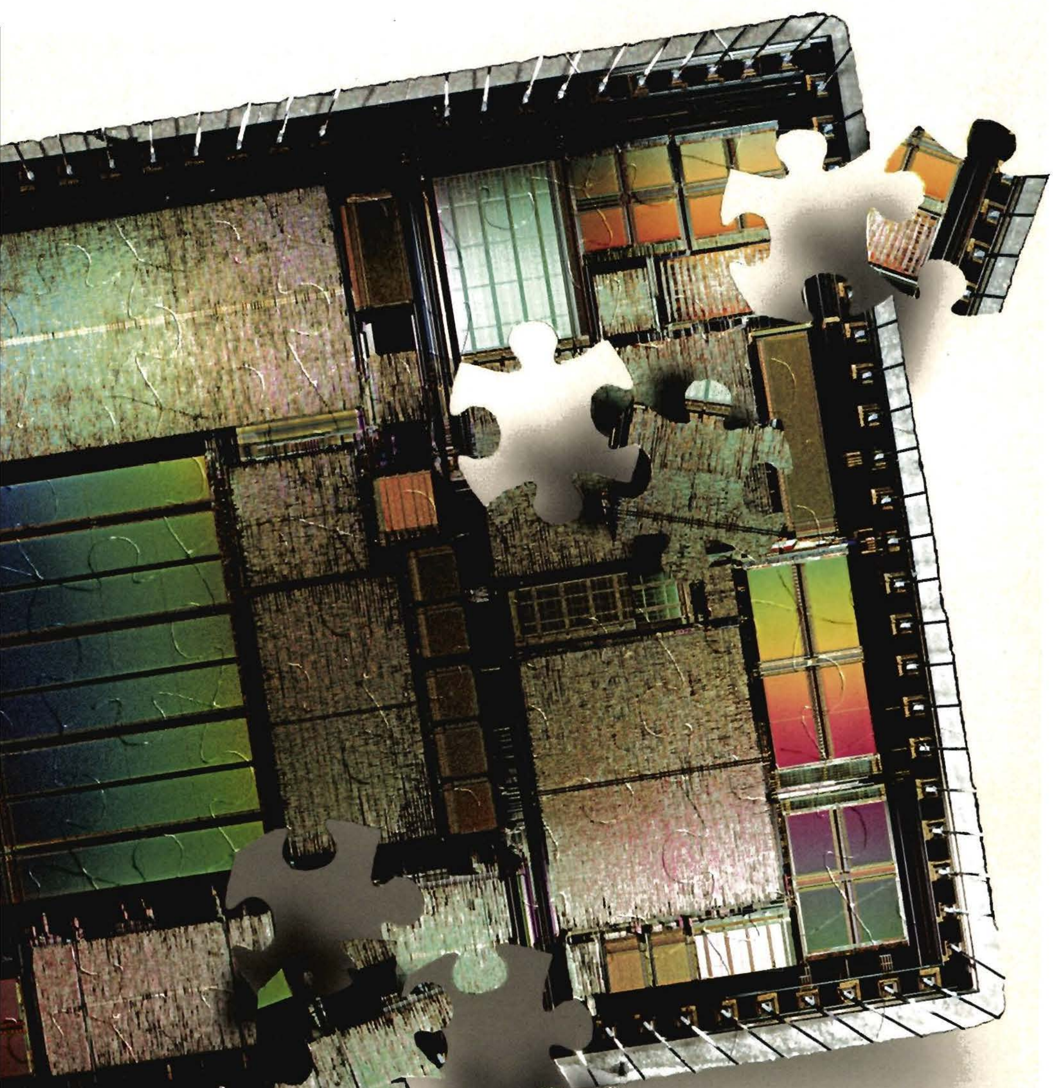
Towards efficient development of complete CAD frameworks

DOI:
10.6100/IR514353

Document status and date:
Published: 01/01/1998

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Towards Efficient Development of Complete CAD Frameworks



Willem Rovers

Towards Efficient Development of Complete CAD Frameworks

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr. M. Rem,
voor een commissie aangewezen door het College
voor Promoties in het openbaar te verdedigen op
dinsdag 29 september 1998 om 16.00 uur

DOOR

Wilhelmus Martinus Henricus Maria Rovers

geboren te Gemert

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. C.J. Koomen
en
prof.ir. M.P.J. Stevens

Copromotor: dr.ir. J.W.G. Fleurkens

Cover: The DABchic IC. Courtesy of the DAB-team from Philips Research

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Rovers, Wilhelmus M.H.M.

Towards efficient development of complete CAD frameworks / by Wilhelmus M.H.M. Rovers. - Eindhoven : Technische Universiteit Eindhoven, 1998.
Proefschrift. - ISBN 90-386-0470-X
NUGI 832
Trefw.: grote geintegreerde schakelingen ; CAD / datamanagement / Petri-netwerken.
Subject headings: integrated circuit design / circuit CAD / Petri nets.

Summary

In this thesis we are concerned with the development of *complete Computer Aided Design frameworks*. CAD frameworks enable designers to create complex systems by managing their design processes. These processes are managed by the design management system component of a CAD framework. In order for a CAD framework to be considered complete it has to provide support for all three design management sub-activities, so for design data, design tool, and design flow management. The design management component of a CAD framework is a dynamic system which is updated regularly, for instance by the addition of new tools or design flows. The persons creating and updating the design management system are referred to as *design management system developers*. Besides supporting the designers, a complete CAD framework should also assist the design management system developers in performing their job. This thesis proposes a method to *efficiently* develop such complete CAD frameworks.

For a long time there has been a lot of confusion regarding what design management is. As a result, it was even less apparent what the structure of CAD frameworks providing this service had to be. Notwithstanding this, a number of systems were made. Most of these systems however, provided only a part of the CAD framework functionality, usually some form of design data management. Some of these systems were extended later on to provide support for the other design management activities as well. Since these systems were originally created with the first activity in mind, this often resulted in non-uniform systems which are difficult to use. In order to design a complete CAD framework it is essential to have a good picture of the complete system. Therefore, this thesis presents a detailed CAD framework model. This model can be used to reason about CAD frameworks and can also serve as a guideline for the implementation of such a system.

The existing CAD frameworks provide hardly any support to the design management system developers. In this thesis the CAD framework model is used to show how this aspect of framework completeness can be implemented. This model points out that their exists an analogy between IC designers and design management system developers, namely that all these people are designers. The only difference between these designers is what they design, namely ICs and design management systems, respectively. Like IC designers, DMS developers produce lots of data and use a variety of tools while creating their DMS descriptions. Therefore, the support provided by the framework to DMS developers should include management of their data, tools and flows. A CAD framework can realise this by making the support facilities it provides to the IC designers available to the DMS developers.

Besides assisting all participants, complete CAD frameworks feature a design management system which provides support for all design management sub-activities. Despite this, the design management systems of most of the existing CAD frameworks do either not support all three activities or lack a good integration of these. The main reason for this is that these systems were directly implemented based on an often incomplete and informal notion about what design management is, how the different design management activities have to be integrated and how to construct a system providing support for these. To avoid these problems, the CAD framework model presented in this thesis features a mathematical description of its design management system. The model is based on an analogy between design management systems and digital systems. This abstract design management system model describes both the behaviour of the system's three main components, i.e., the design data, the design tool and the design flow management systems and their interaction. This model can be used as a guideline for the construction of complete design management systems featuring a seamless integration of design data, design tool and design flow management. The feasibility of this model is demonstrated by using it to represent a design management system for the LOCAM logic synthesis system developed by Philips.

The time required for the development of a CAD framework is almost entirely consumed by the construction and maintenance of its central component, i.e., the design management system. Therefore, the efficiency of CAD framework development can be greatly improved by providing better assistance to the design management system developers. Currently these developers implement their design management systems using a general purpose programming language. Since such a language does not feature special constructs for CAD framework construction, the creation of a design management system will be very time consuming task. Moreover, the resulting system will be difficult to maintain and extend with new functionality. Therefore, the CAD framework development time can be greatly reduced if it is programmed using a dedicated design management system description language. Using such a language, the system under development can be represented at a higher level of abstraction. As far as we know, there are currently no suitable design management system description languages available. In this thesis it is demonstrated how such a language can be defined. Using the presented CAD framework model as a guideline, first the requirements for a design management system description language are formulated. Subsequently *interactive hierarchical coloured Petri nets* are introduced. Furthermore it is demonstrated that languages based on this extension of the hierarchical coloured Petri net concept satisfy the presented requirements. The suitability of such a language is demonstrated by showing how it can be used to create executable design management system representations.

Samenvatting

In dit proefschrift behandelen we de ontwikkeling van *complete Computer Ondersteund Ontwerp (COO) framewerken.* COO-framewerken beheren de ontwerpprocessen van ontwerpers en stellen hen daardoor in staat om complexe systemen te maken. Deze service wordt verleend door de OntwerpBeheerSysteem (OBS) component van een COO-framewerk. Het OBS van een compleet COO-framewerk verleent drie soorten services, namelijk ontwerpdata-, ontwerpprogramma- en ontwerpmethodologiebeheer. Een OBS is een dynamisch systeem dat regelmatig gewijzigd wordt, bijvoorbeeld door de toevoeging van nieuwe ontwerpprogramma's of methodologieën. De personen die ontwerpbeheersystemen maken en/of wijzigen worden *ontwerpbeheersysteemontwikkelaars* genoemd. Naast het assisteren van ontwerpers moet een compleet COO-framewerk ook de werkzaamheden van deze OBS-ontwikkelaars ondersteunen. In dit proefschrift zal een methode geïntroduceerd worden welke gebruikt kan worden om op een efficiënte manier complete COO-framewerken te ontwikkelen.

Er is gedurende een lange periode veel verwarring geweest over wat ontwerpbeheer nu eigenlijk inhoudt. Het gevolg was dat er nog minder duidelijkheid bestond over de structuur van de bijbehorende COO-framewerken. Ondanks dit zijn er toch een aantal van deze systemen gemaakt. De meeste van deze framewerken verleenden echter maar een gedeelte van de gewenste COO-framewerk functionaliteit, veelal een bepaalde vorm van ontwerpdatabeheer. Sommige van deze systemen zijn later uitgebreid met ontwerpprogramma- en/of ontwerpmethodologiebeheer. Doordat deze systemen echter oorspronkelijk ontworpen waren voor ontwerpdatabeheer, zijn de meeste niet-uniform en moeilijk te gebruiken. Om een compleet COO-framewerk te kunnen ontwerpen is het absoluut noodzakelijk om een goed beeld te hebben van het complete systeem. Dit is de reden waarom we in dit proefschrift een gedetailleerd COO-framewerkmodel presenteren. Dit model kan niet alleen gebruikt worden om over COO-framewerken te redeneren, maar geeft ook richtlijnen voor de realisatie van zo'n systeem.

De bestaande COO-framewerken verlenen erg weinig ondersteuning aan de ontwerpbeheersysteemontwikkelaars. In dit proefschrift gebruiken we het COO-framewerkmodel om te laten zien hoe dit aspect van framewerkcompleetheid gerealiseerd kan worden. Het model laat zien dat er een analogie bestaat tussen IC-ontwerpers en OBS-ontwikkelaars, namelijk dat beiden te beschouwen zijn als ontwerpers. Het enige verschil tussen beide typen ontwerpers is wat zij ontwerpen, namelijk respectievelijk IC's en ontwerpbeheersystemen. Net zoals IC-ontwerpers, produceren OBS-ontwikkelaars grote hoeveelheden data en gebruiken

ze allerlei programma's om hun OBS-beschrijvingen te creëren. De ondersteuning welke door het framewerk verleend wordt aan de OBS-ontwikkelaars zou daarom ook het beheer van hun data, programma's en methodologieën moeten inhouden. Dit kan gerealiseerd worden door de ondersteuningsfaciliteiten welk een framewerk aanbiedt aan de IC-ontwerpers ook beschikbaar te maken voor de OBS-ontwikkelaars.

De meeste bestaande COO-framewerken zijn incompleet in de zin dat ze niet alle drie de ontwerpbeheerservices verlenen. Bovendien zijn deze services vaak niet goed geïntegreerd. De belangrijkste oorzaak hiervan is dat deze systemen geïmplementeerd zijn gebaseerd op een vaak incompleet en informeel idee over wat ontwerpbeheer is en hoe het bijbehorende ontwerpbeheersysteem gemaakt zou moeten worden. Om deze problemen te vermijden maakt het in dit proefschrift gepresenteerde COO-framewerkmodel gebruik van een wiskundige beschrijving van het bijbehorende ontwerpbeheersysteem. Deze beschrijving is gebaseerd op een analogie tussen ontwerpbeheersystemen en digitale systemen. Dit leidt tot een abstract ontwerpbeheersysteemmodel welk een beschrijving geeft van zowel het gedrag van de drie OBS componenten, dus van het ontwerpdata-, het ontwerpprogramma-, en het ontwerpmethodologiebeheersysteem, als van hun interactie. Dit model kan dienen als een richtlijn voor de constructie van complete ontwerpbeheersystemen welke de drie ontwerpbeheerservices op een naadloze manier integreren. De bruikbaarheid van dit model wordt gedemonstreerd door het te gebruiken tijdens de beschrijving van een ontwerpbeheersysteem voor het LOCAM logische-synthese systeem ontwikkeld door Philips.

De ontwikkeltijd van een COO-framewerk wordt bijna geheel gebruikt voor de constructie en het onderhoud van het bijbehorende ontwerpbeheersysteem. De efficiëntie van COO-framewerkontwikkeling kan daarom sterk verbeterd worden door de OBS-ontwikkelaars beter te ondersteunen. De huidige ontwerpbeheersystemen zijn geprogrammeerd in een universele programmeertaal. Aangezien dergelijke talen geen speciale constructies voor OBS-ontwikkeling bevatten, zal het programmeren van een ontwerpbeheersysteem in een dergelijke taal een tijdrovend karwei zijn. Een ander nadeel is dat het resulterende systeem zowel moeilijk te onderhouden als uit te breiden zal zijn. De ontwikkeltijd van een COO-framewerk kan daarom sterk gereduceerd worden door het bijbehorende OBS te programmeren in een speciaal ontworpen ontwerpbeheersysteembeschrijvingstaal. In een dergelijke taal kan het te ontwikkelen systeem beschreven worden op een hoger niveau van abstractie. Zover we weten zijn er op het moment echter geen geschikte ontwerpbeheersysteembeschrijvingstalen beschikbaar. In dit proefschrift laten we daarom zien hoe een dergelijke taal gedefinieerd kan worden. Gebruik makend van de richtlijnen verkregen uit het COO-framewerkmodel, definiëren we eerst de eisen waaraan de OBS-beschrijvingstaal moet voldoen. Vervolgens worden *interactieve hiërarchische gekleurde Petri-netten* geïntroduceerd. Daarna laten we zien dat talen gebaseerd op deze uitbreiding van het hiërarchische gekleurde Petri-net concept voldoen aan de gestelde eisen. De bruikbaarheid van een dergelijke taal wordt gedemonstreerd door te laten zien hoe deze gebruikt kan worden om executeerbare ontwerpbeheersysteembeschrijvingen te creëren.

Acknowledgements

I would like to thank prof. dr. ir. C.J. Koomen and prof. ir. M.P.J. Stevens for their support and for given me the opportunity to perform my Ph.D. research. Doing this research enabled me to switch from theoretical physics to electrical engineering and computer science.

I would also like to thank Emile Aarts, Hans Fleurkens, Willem Reijntjes, and Jeroen Voeten for their invaluable comments which helped me to improve the quality of this thesis.

Furthermore I would like to thank Rudi Bloks, Frank Budzelaar, Ton van Doorn, Yun Chao Hu, Robert Huis in 't Veld, Sherif El Kassas, Ardo van Rangelrooij, and Ad Verschueren for their contributions ranging from drinking coffee together to discussions about my research. I'm also grateful to my other colleagues from the ICS group at the Eindhoven University of Technology for making my stay there a very agreeable one. In addition I would like to thank Rian van Gaalen for her excellent assistance.

I'm also grateful to my colleagues at Philips Research in Eindhoven, especially Marc Verra, because he motivated me to finish my thesis by constantly reminding me that it was not finished yet.

Finally, I want to express my gratitude to my parents and my wife. Without their support writing this thesis would not have been possible.

Abbreviations

ALU	Arithmetic Logic Unit
BD	Bus Definition set
CAD	Computer Aided Design
CB	Copying Bus
CD	Compact Disk
CI	Control Interpreter
CPN	Coloured Petri Net
CS	Current State
CTKO	CAD Tool Knowledge Object
CU	Control Unit
DDMS	Design Data Management System
DDS	Design Data Store
DDSIS	DDS Input Sequence
DDSOS	DDS Output Sequence
DF	Delete Function
DFMS	Design Flow Management System
DFS	Design Flow Store
DMS	Design Management System
DTMS	Design Tool Management System
DTMSO	DTMS Output
DTS	Design Tool Store
DTSIS	DTS Input Sequence
DTSO	DTS Output
DTSOS	DTS Output Sequence
EDIF	Electronic Design Interchange Format
ELLA	Electronic Logic Language
FEE	Flow Execution Engine
FId	Flow Identifier
FIFO	First In First Out
HDL	High-level Design Language

IC	Integrated Circuit
IGF	Input Generation Function
IPU	Information Processing Unit
NDL	Network Description Language
NSF	Next State Function
OF	Output Function
OGF	Output Generation Function
OMA	Optimizer and MAtcher tool
OPB	One Place Buffer
OTC	Overspecified Timing Constraints
OTDM	OMA Timing Driven Matching
OTO-D	Object Type Oriented Data
PHACO	Philips HArdware COmpiler
PLANETS	Philips Logic And NETwork Specification format
PTN	Place/Transition Net
RC	Random Colour
RF	Read Function
RS	Random Selection
RTC	Realistic Timing Constraints
RTL	Register Transfer Level
TE	Transition Expression
TEE	Tool Execution Engine
TEEI	TEE Input
TEEO	TEE Output
TES	Tool Encapsulation Specification
TIGF	TEE Input Generation Function
VHDL	VHSIC Hardware Description Language
WF	Write Function
WS	WorkSpace

Contents

Chapter 1

Introduction

In this thesis we are concerned with the development of Computer Aided Design (CAD) frameworks. In literature [tBBvW91], [RW95], [HNSB90], [KGMB94], [vW93], [vW94], [ZG96], there is still a lot of confusion about the nature of CAD frameworks. Despite this, the following facts are agreed upon. A CAD framework should assist designers in managing their design processes. We refer to the CAD framework component which is responsible for this as the *Design Management System* (DMS). This design management system is not a static system, but is updated regularly, for instance by the addition of new tools or design flows. The persons updating the design management system are referred to as *design management system developers.* Besides supporting the designers, a CAD framework should also assist the design management system developers in performing their job.

CAD frameworks enable designers to design complex systems. The creation of complex designs is a complicated and time-consuming task. Designers are still able to do this by making use of the following complexity handling methods: Abstraction and decomposition. The time required for creating complex designs can be kept within realistic boundaries by making use of the following design-time reduction methods: Design reuse, teamwork, and automation. In Section 1.1 these complexity handling and time reduction methods are illustrated.

The use of the previously mentioned complexity management methods comes at a price; designers only benefit from these methods if they are applied correctly. The activity of supporting the designer in making proper use of these methods is referred to as *design management.* Design management is the topic discussed in Section 1.2.

The development of CAD frameworks is an active area of research. Currently, there are no commercially available CAD frameworks that can handle all the aspects of design management. In Section 1.3 the problems associated with currently available CAD frameworks are discussed. Section 1.4 gives an overview of the approach presented in this thesis to solve these problems and of the main results. Finally Section 1.5 gives an outline of the organisation of the remainder of this thesis.

1.1 The design of complex systems

Before we can start to discuss about how CAD frameworks should assist designers in managing their design processes, we first have to know what a design process is. For general models of the IC design process we refer to [All90], [Koo91], and [Sie91]. In [Yos81] and [TY86] the design process is considered from a mechanical engineering viewpoint. [Das89] presents the structure of design processes in general. However, for our discussions about design management, the detailed models presented in these articles are not required; it is sufficient to know how the previously presented complexity management methods enable designers to create complex designs. In this section these methods are introduced. Although these methods can be used for the design of complex systems in general, they will be illustrated using examples taken from the area of Integrated Circuit (IC) design.

1.1.1 Abstraction

ICs are characterised by their behaviour and by properties such as speed, power dissipation and size. Before the manufacturing process of an IC can start, a description of its layout must be created. Due to the complexity of state of the art ICs, it is nearly impossible to design them starting at the layout level. The layout description simply contains too much detail, whilst not explicitly representing the IC behaviour. Therefore, other, more abstract, representations of the IC are used. These descriptions provide a more explicit representation of some aspects of either the IC's behaviour and/or structure.

Figure 1.1 depicts how a multiple abstraction level design process is performed. A design process starts by formulating the requirements, which are a number of statements about the properties the system is *required* to have. After the requirements have been formulated, a first representation of the system is created, which describes the system at the highest level of abstraction. The properties of the system that are explicitly represented at this level of abstraction are *optimised* at that level. When, after a number of *optimisation steps*, these properties satisfy the requirements, the optimised description is transformed using a *detailing step* to a more *detailed* description at the next level of abstraction. This description forms the starting point for the optimisation of the properties specific for this lower abstraction level. This process repeats itself until a description at the lowest level of abstraction is obtained. The multiple abstraction level design process will only produce correct designs if the design representations at the different levels of abstraction are consistent. Detailing and the subsequent optimisations should not invalidate the optimisations made at the higher abstraction levels. Therefore, the optimisations are usually followed by a *verification step* during which it is *verified* that an optimised design representation is consistent with the more abstract representation it was derived from.

Figure 1.2 shows the abstraction levels that are most frequently used during the design of digital ICs ([MPC90]). The first two levels are not specific for IC design, but can be used to represent information processing systems in general. The system level is the top level of abstraction. At this level the general structure of the system is described. Typically the

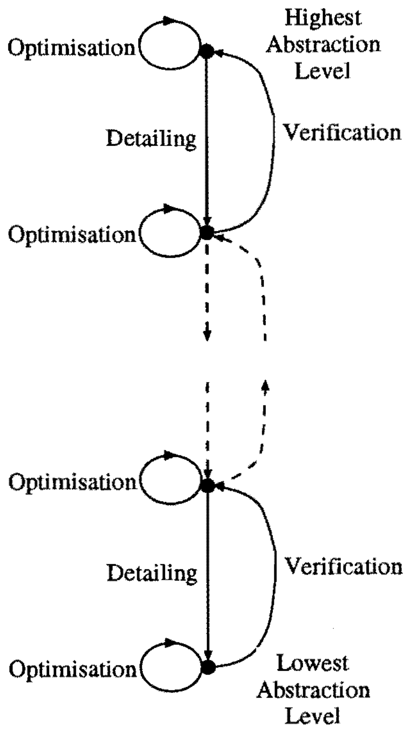


Figure 1.1: A multiple abstraction level design process.

system is described as a composition of a number of related sub-systems, which are either implemented in hardware or in software. At the functional level, an algorithmic description of the behaviour of these sub-systems is given. For the sub-systems implemented in an IC, the next abstraction level will be the Register Transfer Level. At this level the system is described in terms of a number of *registers*, an *output function* and a *next state function*. The registers determine the state of the system. Given the system's current state and input, the output and next state functions determine what the system's output and next state are, respectively. At the logical level, these functions are described in terms of a number of logical expressions. During the detailing step, which transforms a logical level to a gate level description, these expressions are mapped to a collection of interconnected gates, such as AND, OR, NOT, NAND and NOR gates. At the transistor level the interconnected registers and gates are described in terms of a transistor netlist. During the last detailing step, this transistor netlist is transformed into a layout, which implements it in silicon. The layout describes the IC in terms of a number of layers, each representing the areas where a certain type of material should be deposited or removed. The layout description is used to produce the masks employed during the IC fabrication process.

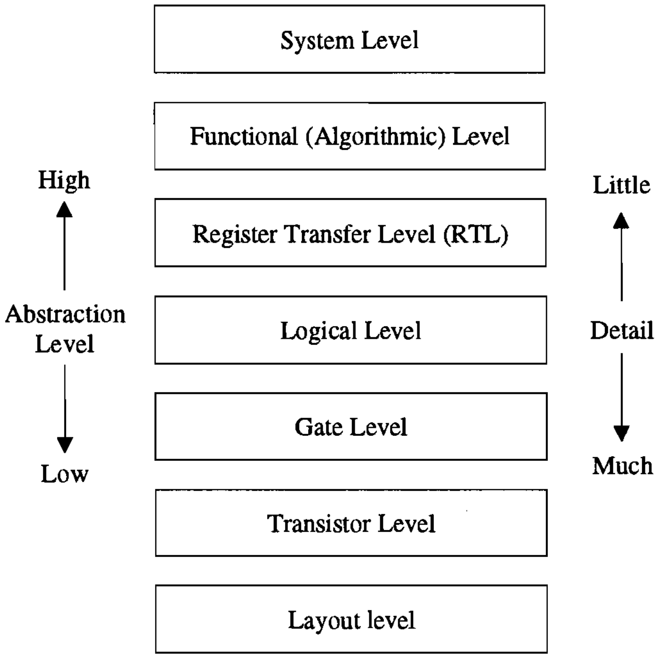


Figure 1.2: Abstraction levels used during IC design.

1.1.2 Decomposition

Another way to handle complex designs is by making use of decomposition [Nie86]. Decomposition reduces the complexity of the design process by splitting up the original design into a number of components, which can then be designed separately. Often these design components are still very complex. Therefore, some of these will again be decomposed. This process of decomposition continues until each component is small enough to be handled conveniently as a single unit. The result of the decomposition process is a hierarchy of design components. In Figure 1.3 the decomposition hierarchy of a microprocessor is shown. The design of this microprocessor has been split into two parts, namely a datapath and a controller. The datapath has again been decomposed into an arithmetic logic unit (ALU), a memory and a multiplier.

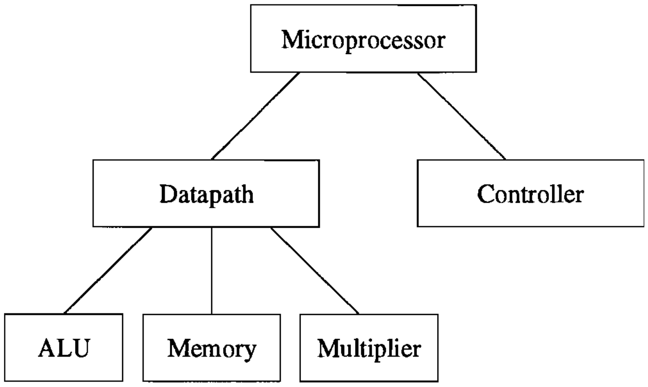


Figure 1.3: A design hierarchy.

1.1.3 Design reuse

An obvious method for reducing the time required to create a design is by reusing other designs. The following three modes of reuse can be distinguished [AOS94]: Reuse by adaptation, reuse by instantiation, and reuse by generation. In the case of *reuse by adaptation* an existing design is adapted such that it satisfies a given requirements specification. For example, if a 15-bit multiplier is required, it can very quickly be obtained by stripping the most significant output from a 16-bit multiplier. RTL synthesis [BRSVW87] [BHMSV84] is a good example of a design process that makes use of *reuse by instantiation.* During RTL synthesis the objects to be reused are pre-designed logic gates and registers stored in a library. Starting from a RTL level design description, the synthesis system will create a functionally equivalent netlist of interconnected library element instantiations. This netlist is converted to a layout by using a placement and a routing program. The placement program determines where the pre-designed pieces of layout representing the instantiated library elements will be put on the layout. The routing program completes the layout by generating the required interconnections. The concept of *reuse by generation* can be illustrated using architecture synthesis [MPC90] [Sto91] as an example. For architecture synthesis the designs to be reused are not simple blocks like gates and registers, but parameterised blocks such as memories or arithmetic logic units. The designer selects which blocks he or she requires in his or her design and determines the desired parameters for these blocks. In contrast to RTL synthesis, the layout of the employed blocks is not taken from a library, but generated using a module generator. In this case, the module generator implicitly represents a complete class of reusable blocks, namely one block for each possible parameter setting.

1.1.4 Teamwork

Teamwork will only result in a reduction of the required design time if the design process can be split up into a number of relatively independent sub-processes, each of which can be performed by one of the team members. When teamwork is combined with the previously introduced complexity and design time reduction methods, this results in a natural division of the total design task.

The use of multiple abstraction levels splits the total design process into a number of sub-processes, namely one for each employed abstraction level. For example, the transistor netlist and the layout implementing this netlist are sometimes created by different people. This means that designers do not have to know all the details of the complete design process, but can become experts on some of the design subtasks.

Decomposition splits a design into a number of smaller design components, which can be developed separately. Except for the interfaces to their own component, designers do not participate in the development of the other design parts.

Design reuse also results in a natural division of the total design task. The first task is the creation of the basic building blocks to be reused, e.g. the libraries and module generators used in RTL and architectural synthesis, respectively. The second task is the maintenance of these basic building blocks. The last task is designing by making use of these building blocks. All these tasks can be performed by different people.

1.1.5 Automation

For certain steps of the design process an algorithm can be found describing how it is performed. These design steps are candidates for being automated. Design process automation means that the design step is performed by a piece of software rather than by the designer. The programs automating such a design step are commonly referred to as *synthesis tools*. Design automation not only reduces the design time, but also enables the designer to carry out a design step without the need to have much knowledge about how it is done. The drawback of design automation is that (partly) automatically generated designs are, in general, less efficient than custom made designs. For many applications however, the performance of the generated designs suffices.

Automation is very common for software design, where programs are written using a high level programming language. For execution of the program, however, a machine language representation is required. These low-level representations are automatically generated from the high level program using a tool referred to as a compiler.

Examples of highly automated design processes in the IC design domain are RTL and architectural synthesis. In the case of RTL synthesis the designer will first create a register transfer level representation of his design. After the behaviour described by this design representation is validated using simulation, the rest of the design process is completely automated. In the first step of this process the RTL description is converted into a more detailed logical level design representation. Subsequently the number of expressions in this design representation is reduced using a tool referred to as a logic optimise. The resulting description is mapped by a matcher tool onto a network of interconnected gate level elements, i.e. gates and registers. The corresponding layout is created by a placement and routing tool. In the placement step, the pre-designed pieces of layout of the employed gate level elements are put at a certain location in the layout. In the routing step the layout is completed by adding new pieces of layout, which implement the connections of the gate level network.

1.2 Design management

Design management is a combination of three sub-activities, namely design data, design tool, and design flow management [SD96]. These three activities are the topic of the remainder of this section.

Design data management is the activity of keeping track of all the design data and their relations and the use of these relations for controlling the access to data by multiple designers. The need for design data management is directly related to the use of the complexity handling and time reduction methods introduced above. These methods not only enable designers to create complex designs within a reasonable amount of time, but also give rise to substantial overhead. Instead of one description of the IC, there is a collection of design components, each represented at multiple levels of abstraction. The situation is worsened by the fact that during the design process many different versions of these design parts are created. Unless the designer is able to keep track of the relations between all these pieces of data, abstraction and decomposition will be useless. Libraries are only useful if the designers are able to find the desired library elements. When a team of designers is working on a design the situation becomes even more complex. In this case a data access control mechanism is required, preventing designers from making mistakes such as overwriting other designers' data or using the wrong version of another person's design part in a simulation.

Design tool management is the activity of assisting the designer in making proper use of the available tools. Although design automation tools potentially reduce the design time, a real reduction is only achieved if the designers know how to use these tools. For instance, the designers have to know what a certain tool can be used for, how the tool is invoked and how to fine-tune the tool for their specific application. A design tool management system will typically abstract from tool invocation related information and will provide its users with tool characterisation information to be used for tool selection.

If the designer's data and tools are managed, the only task remaining for the designer is to decide how the required design can be obtained using these tools and data. This involves selection of the tools to invoke, determination of the order of these invocations and selection of the data to which these tools are applied. Although the terms *design methodology management* [HNSB90] [KGMB94] and *design process management* [JD92]

[SD96] are frequently encountered in literature, we refer to the activity of assisting the designer in making these decisions as *design flow management* [tB95] [vW93] [vW94].

1.3 Problem definition

Development of a CAD framework is not a trivial task; it usually takes a lot of painstaking work to create such a system. This is especially true when, like in the case of most of the currently available CAD frameworks, such a system has to be build from scratch using a general purpose programming language, which does not feature special constructs for CAD framework construction. In addition to the fact that their construction requires a lot of effort, the resulting systems will often be difficult to maintain and/or to extend with new functionality. Therefore it would be very useful to have a method enabling us to develop CAD frameworks more efficiently.

Another problem associated with the currently available CAD frameworks is that most of these are not complete; they either do not provide support for all three design management sub-activities and/or they provide little or no assistance to the design management system developers. Typically, CAD frameworks originated as a design data management system [vW93] [vW94]. Despite the difficulties encountered, some of these systems were then extended to provide support for the design tool and design flow management activities as well. However, due to the fact that these systems were originally designed with design data management in mind, it was not always possible to achieve a good integration of the different design management system components. So even though some CAD frameworks in principle support all three activities, these systems will often be non-uniform and difficult to use. The main reason why CAD frameworks are created is to improve the efficiency of designers. Realisation of a framework doing this is already a very difficult task. Therefore, less urgent features such as design management system developer support received little or no attention during the development of the currently available CAD frameworks. As a consequence most of these systems are very designer oriented and provide only poor assistance to the design management system developers.

1.4 Towards a solution

In thesis we propose a method to solve the problems described above, i.e., a method enabling us to *efficiently* develop *complete* CAD frameworks.

One of the characteristics of a complete CAD framework is that it provides support and uniform interfaces to all participants, so to both the designers and design management system developers. In this thesis we will show how this aspect of framework completeness can be implemented. Our approach is based on an analogy between IC designers and design management system developers, which stresses that all these people are designers, who only differ in what they design, namely ICs and design management systems, respectively. Like IC designers, DMS developers produce lots of data and use a variety of tools while creating their DMS descriptions. Therefore, the support provided by the framework to DMS developers should include management of their data, tools and flows. A CAD framework can realise this by making the support facilities it provides to the IC designers available to the DMS developers. These facilities are provided by the design management system component of the CAD framework, so design management system developers will be assisted by the same system they are helping to develop.

When the CAD framework's design management system provides support to all participants, it becomes the central component of the CAD framework. In the past there has been only little consensus about what the structure of such a system should be. Therefore, the design management systems of most of the existing CAD frameworks were directly implemented based on some informal and often incomplete description of this system. This is the main reason for the incompleteness and lack of uniformity of these systems. To avoid these problems we will create a mathematical description of the design management system. This abstract design management system model shows how a good integration of design data, design tool and design flow management can be achieved. This model is based on the analogy existing between design management systems and digital systems.

The time required for the development of a CAD framework is mainly consumed by the construction of its central component; the design management system. The design time of this system can be greatly reduced if it is not programmed in a general purpose language, but instead, is implemented using a dedicated design management system description language, which features special constructs for CAD framework construction. Using such a language, a design management system can be represented at a higher level of abstraction. As a result, it not only becomes easier to create design management system descriptions, but also to maintain and extend these and will therefore lead to a big improvement of the CAD framework development efficiency. However, currently there is no suitable design management system description language available. Therefore, in this thesis we will demonstrate how such a language can be defined.

1.5 Thesis organisation

In Chapter 2 we present an overview of the state of the art for CAD frameworks. This chapter introduces the current ideas about what a CAD framework is, presents some of the techniques developed for design management and shows how these techniques have been integrated in the existing CAD frameworks.

In Chapter 3 up to and including Chapter 6 a CAD framework model is presented, which clearly demonstrates how a CAD framework can be constructed which not only achieves a good integration of design data, design tool and design flow management, but also provides support to all participants.

Chapter 3 introduces the IC designer - DMS developer analogy and shows how this analogy leads to a CAD framework in which the design management system is the central component. In addition it features a model of the design management system based on an analogy with digital systems. This model is illustrated using a design management system for the LOCAM logic synthesis system. The LOCAM system is described in this chapter and will be used throughout this thesis to illustrate the concepts we introduce.

In Chapter 3 the design management system is described in terms of three interacting sub-systems. In the following three chapters, more detailed models of these three sub-systems will be presented, where Chapter 4 covers the design data management system, Chapter 5 the design tool management system and Chapter 6 the design flow management system.

Chapter 7 demonstrates how to define a suitable design management system description language. First the requirements for such a language are formulated. To do this it is essential to have very good picture of what a design management system is. Fortunately, we have already presented a detailed design management model in the previous chapters, which can be used as a guideline. Finally we introduce a class of languages based on an extension of hierarchical coloured Petri nets and demonstrate that these satisfy the formulated requirements.

Finally in Chapter 8 we present our conclusions and we discuss the topics to be addressed by future research.

Chapter 2

CAD frameworks: State of the art

In this chapter an overview of the state of the art for CAD frameworks is given. Besides for giving information about the completeness of the currently available CAD frameworks, this overview will also be used to informally introduce the concepts involved.

Section 2.1 presents the current ideas about what a CAD framework is and about what it is supposed to do. In order to be able to discuss the kind of support a CAD framework has to provide, we have to know what people are involved. Section 2.2 introduces these CAD framework participants. Likewise, discussions about how to integrate the different design management activities only make sense if we know what design data, design tool and design flow management are. Section 2.3 presents the current ideas about these three design management activities. In Section 2.4 we demonstrate how these ideas have been implemented in some of the existing CAD frameworks. Section 2.5 discusses the current ideas regarding the architecture of CAD frameworks. The architectures presented in this section describe a CAD framework in terms of a number of interconnected components, which either assist designers by performing one of the three design management activities or provide support to the other participants. Therefore, Section 2.5 will also present some of the ideas regarding the integration of the three design management activities and the kind of support which should be provided to the CAD framework participants.

2.1 CAD framework definition

Like stated in [KGMB94], defining the term CAD framework is a difficult task. Although the term CAD framework is not exactly defined, the following definitions roughly cover the contents. In [CFI90] the following definition is encountered: "A CAD framework is a software infrastructure that provides a common operating environment for CAD tools." In [tBBvW91] a CAD framework is defined as: "A CAD framework serves as a basis for CAD tool integration and provides the designer with assistance for data organisation and design management." In [HNSB90] the following alternative definition is found: "The term CAD framework has come to mean all of the underlying facilities provided to the CAD tool developer, the CAD system integrator, and the end-user (IC or system designer) necessary to facilitate their tasks." Finally in [KGMB94] the following statement is encountered: "Typically, CAD frameworks include some form of design data management, a consistent user interface, and inter tool communication, in addition to design methodology management."

Despite the confusion regarding the definition of what a CAD framework is, the following things are clear. Firstly, a CAD framework should provide the designer with support for all three design management activities, so for design data, design tool and design flow management. Secondly, besides supporting the designer, the CAD framework should also assist the design management system developers, such as the tool developers and the CAD system integrators mentioned above, in performing their jobs. Therefore, we adopt the following definition: "A CAD framework is a software infrastructure which assists designers by managing their design data, design tools and design flows and which provides support to the design management system developers maintaining it."

Providing support to the Design Management System (DMS) developers is an essential characteristic of CAD frameworks. Many CAD frameworks provide only little assistance to these DMS developers. An extreme example of this is the type of design support system referred to as a *design environment*. Design environments consist of a number of tightly integrated tools operating on a common design data representation. Addition of new tools or alteration of the built in design flow is nearly impossible in these systems. Although a design environment provides design management support, it is usually not considered to be a CAD framework [Dan89], because it completely lacks DMS developer support.

2.2 CAD framework participants

In addition to the confusion about what a CAD framework is, there is also little agreement about how to classify the participants involved. In [HNSB90] the CAD tool developer, the CAD system integrator, and the end-user (IC or system designer) are introduced as the CAD framework participants. In [KGMB94] design tool and flow management are referred to as activities involving the following groups of users:

- Designers - the ultimate end-users, people who produce design data by executing CAD tools in a design flow,
- Tool Developers - people who write CAD programs,
- Tool Integrators - people who combine tools and hide the dependencies between tools. They also customise vendor tools to tailor them to the needs of a specific site,
- Flow Developers - design experts who describe the design methodology to be used, and
- Managers - those who supervise all of the above.

To make the situation even more complex, [LJ92] introduces framework administrators, design methodology managers, project managers and design engineers as the CAD framework participants.

We introduce two types of participants: *Designers* and *design management system developers*. The DMS developers can again be divided into five groups: Tool developers, flow developers, design data management system developers, integrators, and CAD framework managers. As their names suggest, *tool developers* and *flow developers* create the design management system's tools and flows, respectively. Design data management system developers structure the design data management system by defining a useful organisation for the design data managed by this system. Normally tool developers create tools independent from the CAD frameworks in which these will potentially be used. Therefore, in order for a tool to be used in a CAD framework, the tool will have to be integrated with it. This is the task of the *integrators*. In contrast to tools, flows will usually be developed using the flow description language provided by the CAD framework. In this case no integration is required. A CAD framework stores its data in its local database and/or in external databases. In the last case these external databases will have to be integrated with the CAD framework by the integrators. The last participants we introduce are the *CAD framework managers*. It is the task of these managers to ensure that the framework created by the other DMS developers functions correctly. One of their tasks is to add new users to the CAD framework and to grant them with the privileges required for performing their tasks.

2.3 The design management activities

Discussions about how to integrate the different design management activities only make sense if we know what these activities encompass. In the following three sub-sections we introduce the current ideas about design data, design tool and design flow management, respectively.

2.3.1 Design data management

Design data management is concerned with two types of design data, i.e.,

1. raw data, and
2. meta data.

Raw data are the design descriptions and the auxiliary data, such as simulation and analysis results, used and/or produced during the design process. These often large amounts of data are stored in files residing at some location on a computer network. A design data management system should provide the designer with logical rather than physical access to these data. This means that the system should abstract from information about the actual location of the data on the network and of the operating system specific methods used to store and retrieve these data.

Meta data is information about the design representations and the design activities. A design data management system uses meta data to represent raw data properties and to keep track of the relations existing between the different pieces of raw data. A design data management system collects the meta data by monitoring the design process and uses these data to perform its services.

In [RRvHK93] [vHL96] design data management is described as a multi-dimensional problem. In Table 2.1 the six dimensions presented in [RRvHK93] are listed.

Table 2.1: Design data management dimensions.

Dimension	Representation of:	Why?	Example
Versions	Modifications	Keep history	Optimisation
Hierarchy	(De)composition	Divide and Conquer	Subcircuits
Views	Other representation	Abstraction	Layout, schematic
Derivation	Input $\rightarrow$ Output	Tool flow	Simulation results
Workspaces	Team-work	Access control	Library
Variants	Product options	Parallel developments	Mains or battery supply

During the design process many different *versions* of a design representation are created. The reason is that design representations are almost never "first time right". They either contain functional errors or do not satisfy the formulated requirements. A correct design representation is usually obtained after a number of modifications of the original description, each time resulting in a new version. A *hierarchy* represents how the different components of a decomposed design are related. During the design process a number of different *views* of the design are created, each uniquely characterised by the abstraction level at which the design is described and/or by the representation language employed. *Derivation* is concerned with the management of tool invocation results. This is done by association of the resulting outputs with the inputs they were derived from. For example, derivation relates the results of a simulation run to the corresponding design representation and input vectors. For managing team-work, design data management systems employ *workspaces*, which provide a means to control access to data. A product may have different *variants*, which, although they provide almost identical functionality, have different implementations, because they have to operate under different circumstances. For example, CD-players are produced in three variants, namely a home, a car, and a portable version. Because the portable version has to run on a battery supply, the ICs used in this variant have to be optimised with respect to power consumption. However, because the remaining functionality is almost identical for all three applications, the design processes of all three variants will have a large part in common. The design data management system has to keep track of to which variant a piece of data belongs.

It is not difficult to build a design data management system which handles one or more dimensions separately. However, creation of a system which is able to deal with more dimensions simultaneously, without causing data inconsistencies, is not a trivial task. Most of the existing systems only support a few dimensions and have difficulties taking the interaction between these dimensions into account. For a summary of how some of the existing design data management systems handle multiple dimensions we again refer to [RRvHK93].

Most of the existing design data management systems are based on an approach similar to the one introduced by Randy Katz [KBC+87], who proposes an organisation of the design data space based on design objects. He introduces the following two classes of design objects.

Representation objects are associated with raw data *representing* a design part at some level of abstraction. These objects are uniquely identified by their name, version number and type. For example the second version of a layout representation of a datapath is referred to as "Datapath[2].Layout". Representation objects are either *composite* or *primitive.* Primitive objects are associated with a complete representation of a design part. Composite objects on the other hand, correspond to representations describing a design part in terms of how it connects the subsystems it is composed of. Since such a representation does not contain the descriptions of these subsystems, a composite object only provides an incomplete description of the corresponding design part.

Structural objects are system maintained objects introduced with the purpose of introducing a useful structure on the design database. Structural objects can be used to create an explicit representation of design data management dimension related information. Having such an explicit representation is essential for each design data management system.

Katz represents objects and their relations using directed graphs. Nodes represent objects and the labelled edges the relations between these objects. As an example consider the datapath layout version history graph shown in Figure 2.1. Such a graph can be used to represent information about versions, derivations and variants. The top node of this graph represent the "Datapath.Layout" structural object. This object, referred to as a version object, groups all the versions of the layout level datapath representation produced during the design process and can be used to represent information which all these versions have in common. The version history graph associates the datapath layout versions to the corresponding version object "Datapath.Layout" using the *is-a-version-of* relationship. The version history is represented using the *is-a-derivative-of* relationship, which relates a version to the design representation it was derived from. Design variants are often functionally equivalent and only differ by the way they are optimised. For example, the ICs used in a portable product will usually be optimised for power, whereas products which are not powered by a battery are most likely to be optimised for speed and/or area. This results in situations like that for the "Datapath.Layout" versions "1" and "2", which are derived from the same ancestor object, possibly by the same tool but using different settings.

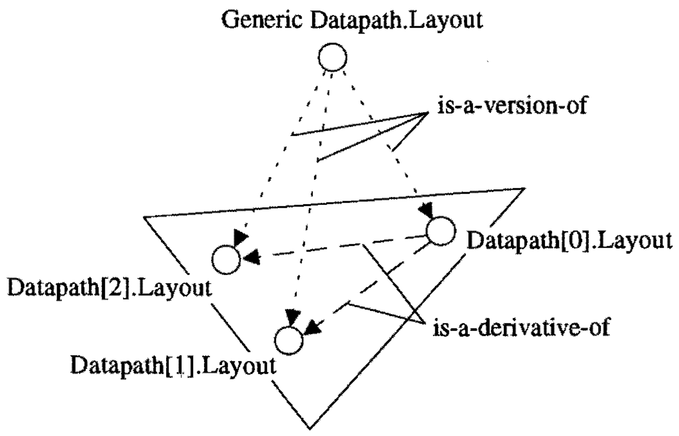


Figure 2.1: Datapath layout version history.

Katz represents information about hierarchy and views using declaration graphs like the one shown in Figure 2.2. The top node of this graph represents an equivalence object, which is a structural object combining all the different views of the Datapath design. The equivalence object combines sub-graphs, referred to as representation hierarchies, which represent the structure of the design representations at a certain level of abstraction. For example, the layout representation hierarchy declares the layout level datapath representation objects as being composite objects defined in terms of an "ALU", "Rfile" (register file) and a "Shifter" subsystem.

To manage the interaction between hierarchy and versions Katz introduces configuration graphs. Using the different versions of the design part representations created during a design process, many versions of the total design representation can be created. Such combinations are called configurations and are created by relating a version of a composite representation object to versions of its components. An example of a graph representing a configuration is presented in Figure 2.3, which combines version "2" of the composite "Datapath.Layout" object with version "1", "1" and "3" of the primitive objects "Rfile.Layout", "Shifter.Layout" and "ALU.Layout", respectively.

For managing team-work Katz uses the workspace concept. He defines a workspace as a named collection of design objects. There are several types of workspaces, characterised by their access rights (check-out/export privileges) and how severe objects are verified before they can be placed into a workspace (check-in/import restrictions). Examples of the different kinds of workspaces employed are the archive, group, and private workspaces. Archive workspaces, also referred to as library workspaces, contain information which is frequently used by lots of designers. Therefore, this information has to be correct and

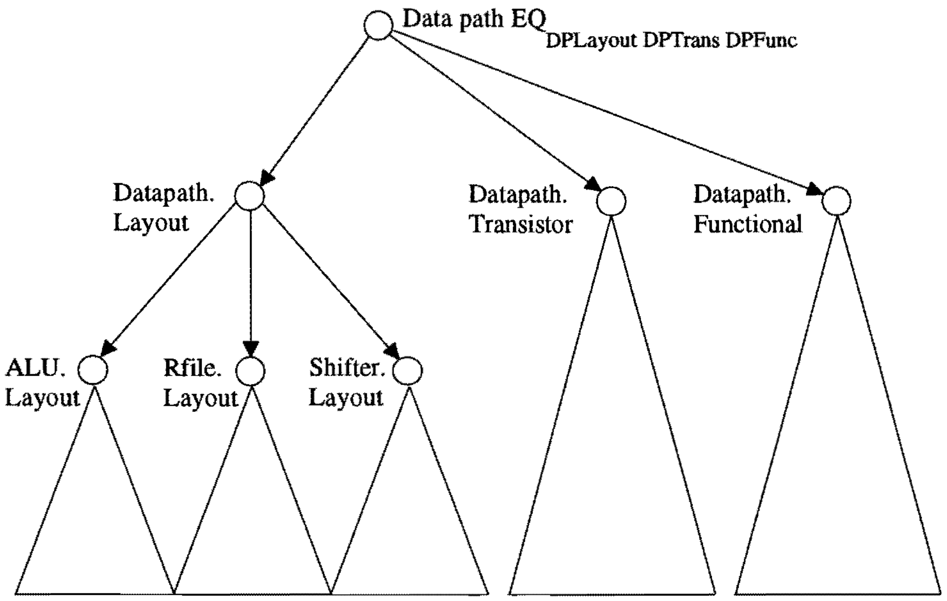


Figure 2.2: A hierarchy and view declaration graph.

accessible to all designers. This is achieved by nearly unrestricted check-outs and very restrictive check-ins. Private workspaces are owned by only one designer. Objects can only be checked out by the designer who owns such a workspace. Of the three types of workspaces presented here, private workspaces have the most restrictive check-outs and the most liberal check-ins. Group workspaces are owned by the members of a design group. In a group workspace the design part representations made by the group members are combined to form the complete design representation. The check-in and check-out restrictions of such a group fall somewhere between those of archive and private workspaces.

Workspaces can be organised in the form of a *workspace hierarchy.* A workspace hierarchy describes the communication structure of its workspaces, i.e., workspaces can only export to and import from workspaces they are related to by the workspace hierarchy.

Figure 2.4 shows a workspace hierarchy employed during the design of the microprocessor of Figure 1.3. It represents that the microprocessor is created by a team consisting of the designers "des1", "des2", "des3" and "des4". The design of the datapath is a joint activity of "des1" and "des2". These designers make their datapath representations in their private workspaces "des1WS" and "des2WS". Their work is co-ordinated using the group workspace "datapathWS", which contains data shared by both designers. The datapath is designed using the information stored in the library workspace "li-

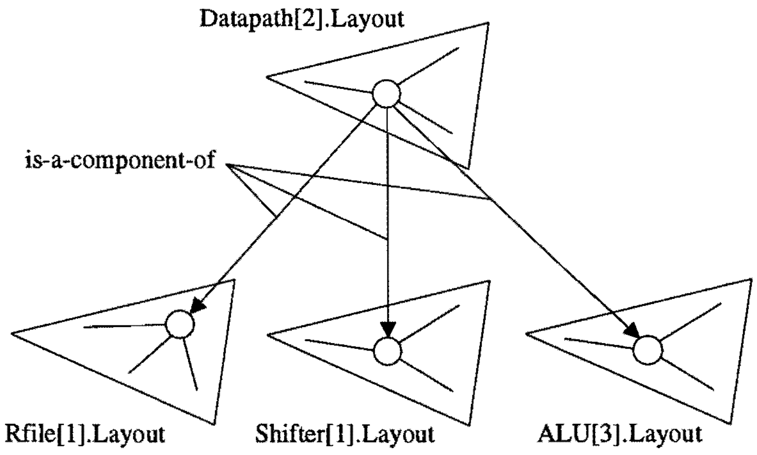


Figure 2.3: Datapath layout configuration.

braryWS". Designers "des3" and "des4" are responsible for the design of the controller and the top-level microprocessor description. Their co-operation is co-ordinated using group workspace "micconWS". The work performed in the datapath and the microprocessor/controller workspaces is co-ordinated using the project workspace "μPProjectWS". A *project workspace* [RRvHK93] is a higher-level group workspace which co-ordinates the work performed in a number of group workspaces.

Usually the workspace hierarchy reflects how the total design task is split into a number of smaller design sub-tasks. In our example the decomposition of the microprocessor design task into a datapath design task and a controller design task formed the basis for the workspace hierarchy. However, the workspace hierarchy can also be based on any of the other ways to divide the design task, e.g., according to abstraction level or, in the case of design reuse, the type of work performed.

2.3.2 Design tool management

As the name suggests, a *design tool management system* manages the tools of a CAD framework. A *tool* is an executable program used to support the designer in his design tasks [LJ92] [vW93] [vW94]. Design tool management can be divided into three sub-activities: Tool integration, tool characterisation, and tool invocation.

Tool integration [CFI90] [FBM94] [HWS92] [Sim93] deals with the addition of new tools to the CAD framework. Design tools are usually developed as stand-alone programs, which are independent from the CAD frameworks in which they are used. Therefore, for most of the tools it will not be possible to directly interface them to the CAD framework. Some

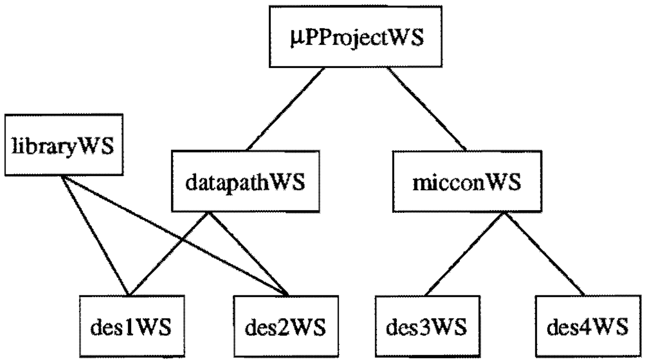


Figure 2.4: A workspace hierarchy.

tools are integrated by modification of the source code, allowing these tools, referred to as *integrated tools*, to directly access the services provided by the CAD framework. However, the source code of a tool is not always available. In this case, the tool is integrated using *encapsulation* [Sch94] [Sch95], i.e., a special program is written, referred to as a *wrapper*, which interfaces the *encapsulated tool* to the CAD framework. To facilitate encapsulation the Tool Encapsulation Specification language TES [CFI91] was developed. This language enables tool developers to the specify tool encapsulation information required by tool integrators for writing a tool wrapper. To support the integration of tools, a CAD framework should feature a *tool interface*, consisting of a number of functions via which the tools can access the services provided by the framework. In [vW93] and [HWS92] two proposals for such a tool interface are presented.

As stated in [DD89], a designer should be able to query the design tool management system about the existence of tools suitable for a given task. This is only possible if the tools are able to accurately represent their general abilities to their potential users. The activity of charting the abilities of a tool is referred to as *tool characterisation*. Except for [DD89], there is not much literature about this topic, probably because it strongly depends on the type of tool being characterised.

Once a designer has selected a tool, he should not be bothered with details about how to invoke the tool. This means that the system should abstract from *tool invocation* related information such as:

- The location of the corresponding executable,
- the parameters to be supplied to this program during invocation, and
- the location of the inputs and outputs of the program.

Like for tool characterisation, [DD89] is about the only article addressing this problem. The reason is probably that tool invocation deals with low level operating system specific issues.

Tool integration, tool characterisation and representation of tool invocation related information are activities performed by the DMS integrator referred to as a *tool integrator*. Sometimes the tool integrator is assisted by the developer of the tool.

2.3.3 Design flow management

Besides managing their data and tools, a CAD framework should also assist designers in applying the tools to the data in such a way that the desired design is obtained. This involves selection of the tools to invoke, determination of the order of these invocations and selection of the data to which these tools are applied. A description of how to make these decisions is referred to as a *flow*. The activity of assisting the designer in making these decisions is referred to as *design flow management*. For a good overview of the existing approaches for design flow management we refer to [tB95] and [KGMB94].

A design flow management system should provide a language to represent flows. Using this language, design flow developers either create an explicit flow representation [vHT90] [vW93] [vW94] [tB95] or a description of a flow generator [BC94] [BC95] [SBD93], which can be used to dynamically define flows. The knowledge required for constructing such a flow is obtained by consulting design experts. The resulting flow representations can be used to document design processes. However, if the flow description language is executable, then these representations can be used to automate the part of he design process the flow describes. In fact, such a flow acts as a higher level tool, to be used by a designer instead of the tools invoked by the flow. This enables managers to enforce a design policy simply by selecting the tools and flows which a designer is allowed to use.

[KGMB94] uses *directed graphs* as a intuitive metaphor to discuss flows. A directed graph is composed of a set of vertices or nodes, a set of edges or arcs, and a mapping of every edge onto an ordered pair of vertices. A design flow can now be represented by a graph, where each node represents a design activity, such as the application of a CAD tool, and each arc describes a dependency, temporal, data or control, the destination node has on the source node.

Flows describe decisions regarding which tools are to be invoked on what data. If a flow directly refers to the tools to be executed, changes to the set of available tools will require some of the flows to be updated. This makes the flows descriptions very volatile. A possible solution is to have the flow refer to tool abstractions like *tasks* [BD91] [HD96] [KGMB94] or *activities* [LJ92] [BtBvW92] instead. These tool abstractions correspond to abstract design functions and abstract from information about how these are implemented in terms of tool invocations. When flows are defined in terms of tool abstractions, then changes to the collection of available tools will only affect the tool abstraction definitions and will leave the flows unchanged.

2.4 Existing CAD frameworks

In this section we will demonstrate how the ideas about the three design management activities presented in the previous section have been implemented in existing CAD frameworks. As an example of how design data management can be implemented we present the Nelsis CAD framework [vWSBD90] [vW93] [vW94]. Tool management is illustrated using the Cadweld framework [DD89], which features an object-oriented approach towards tool management. There exist a large number of systems implementing a certain form of design flow management. [KGMB94] tries to establish a classification of these systems. We illustrate design flow management again using Cadweld [DD89] and Nelsis [tBBvW91] [BtBvW92], because these are good representatives of the blackboard and the data flow approach of design flow management, respectively. Other systems, e.g., Monitor [Jan86] and Decor [KT92], use some sort of Petri Net to represent design flows. Since these formalisms are very similar to the dataflow formalism, we will not treat these systems here.

2.4.1 Nelsis

Nelsis was developed at the Delft University of Technology and is one of the most complete existing CAD frameworks. In contrast to Katz, who uses an informal approach, Nelsis uses the OTO-D data model to formally structure the design data management information. OTO-D [tB92] is an acronym for Object Type Oriented Data model. OTO-D is a powerful semantic data model, which can be used to define object types and their relations. This has resulted in the OTO-D data schema shown in Figure 2.5, which forms the basis of Nelsis' design data management system.

The OTO-D data model defines object types in terms of their attributes. Figure 2.5 shows how an OTO-D data schema can be represented graphically. In this figure boxes are used to represent object types. An object type is related to its attributes by lines running from the bottom of the corresponding box to the top of the boxes representing its attribute types.

The central type in the Nelsis data schema of Figure 2.5 is the "Design Object". The design object is the equivalent of the representation objects used by Katz, because it has a design description associated with it. Design objects belong to a certain module. The "Module" type is the equivalent of Katz's version object, because it is used to group the different versions of a certain design part representation. In order to represent the relation between a composite representation object and the design part representations it is composed of, the "Hierarchy" type has been introduced, which relates a Father-DesignObject (F) to a Sun-DesignObject (S). The "Equivalence" type is used to related an Original-DesignObject (O) to a Derived-DesignObject (D) together with the "Tool" used to perform the "Transaction". Finally, the object types "ImportedDO" and "Export" are used to model the workspace mechanism.

Nelsis originated as a design data management system. Later on, a design flow management system was added to it [tBBvW91] [BtBvW92] [tB95], which uses data flow graphs for

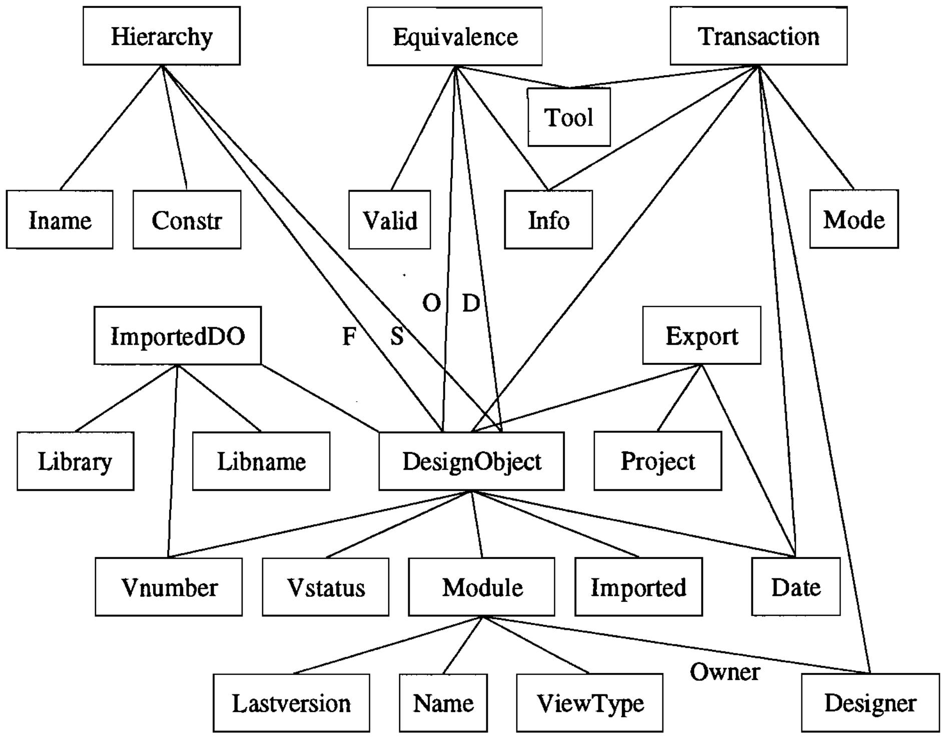


Figure 2.5: OTO-D data schema used by Nelsis.

design flow representation.

One of the difficulties flow management systems have to face is the fact that tool behaviour may vary due to several external variables, such as command line arguments (options and parameters), control files and user interaction. Due to these variables the number and type of the input and output data consumed and produced during a tool run may vary. If this is the case, it is impossible to check whether all input data necessary for running a tool are available. It also disables the possibility to check for tool termination by monitoring the produced output data. Because of this, design flow management becomes nearly impossible. This problem can be tackled by using tool abstractions referred to as activities in flow descriptions rather than the tools themselves. Nelsis creates activities by combining a tool with assumptions about its external variables, such that its input and output files are fixed. For each different input/output configuration of a tool an activity is defined. However, it is also possible to create multiple activities having the same input/output configuration. In general, a tool will be split into that set of activities representing the different design functions of a tool.

As stated before, Nelsis uses data flow graphs for design flow management. An example of such a graph is given in Figure 2.6. Nelsis' data flow graphs consist of a number of interconnected functional units. These functional units, denoted by rectangles, can be used to model activities. By additionally allowing these units to represent subdataflows, like the Extract sub dataflow in Figure 2.6, hierarchy is introduced in the data flow graphs. The information transfer between the functional units of a data flow graph is represented using ports, denoted by the diamonds, circles and squares at the edges of the functional units, and channels which are graphically depicted using arrows. A port explicitly represents the presence of an input or output of a certain datatype for the corresponding functional unit. Ports are connected by channels. The arrows depicting the channels point from the output ports, at which the produced data are available, to input ports of the same datatype, which consume these data.

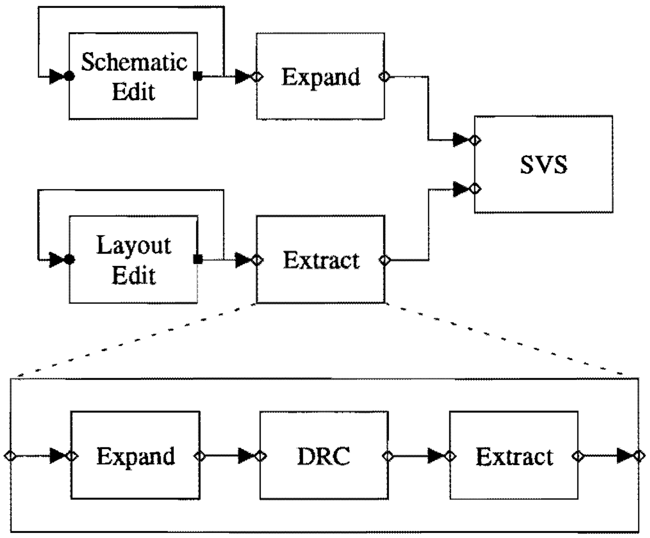


Figure 2.6: Layout Versus Schematic dataflow.

Nelsis uses these data flow graphs in the following way to perform design flow management. Activities represented by functional units with no input ports can always be executed. The output data produced by such an activity are transferred from the output ports via the channels to the connected input ports. After this, those activities which have data on all their input ports, can be executed. The termination of an activity can be determined by checking if data have been produced for all its output ports. The generated outputs are again transferred to the corresponding input ports, thus enabling other activities to be activated.

The input and output ports of functional units are again subdivided into two different types. Activities can only be executed if data of the correct type are available on the input ports. However, for some activities certain inputs are allowed but not required for their proper execution. To represent these kind of inputs Nelsis introduces optional ports (denoted by solid circles), which in contrast to the normal mandatory input ports (denoted by diamonds), do not have to contain data at the moment the corresponding activity is activated. In order to support versioning Nelsis uses two types of output ports. For the ports denoted by diamonds, also referred to as extension ports, the new data which become available for this port overwrite the existing data. As an alternative for this, modification ports are introduced (represented by solid squares), for which the new data does not invalidate the old data present at this port.

In Figure 2.6 the flow of a Layout Versus Schematic (LVS) verification is represented. The schematic and the layout are created using an editor. Schematics and layouts generated using these editors are stored at the corresponding modification output port. The optional input port denotes that the editors either create a new design representation (no input) or a modification of an existing one (input taken from the editor's output port). When both a schematic and the layout implementing it are present, a Schematic Versus Schematic (SVS) consistency check can be performed. This verification is done by comparison of the flat schematic (created by Expand) and the flat schematic extracted from the layout. The Extract sub flow describes the sequence of steps involved in this extraction: A layout expansion, a Design Rule Check (DRC), and a netlist extraction.

2.4.2 Cadweld

Cadweld is a CAD framework developed at the Carnegie Mellon University by James Daniell and Stephen W. Director [DD89] [Dan89]. It features an object oriented approach to tool management and a blackboard mechanism for design flow management. The general structure of Cadweld is shown in Figure 2.7.

Like Ulysses [BD89], from which it inherited some of its features, Cadweld uses a blackboard mechanism to support communication between the designers and the CAD tools. If a designer wants to perform a certain task, he posts the description of this task on the blackboard. In reaction to this, some tools volunteer for a chance to activate. Using the description of the abilities of these tools, the designer will pick one of the tools to perform the task. After its completion, the tool will post the results on the blackboard for examination by the designer. After this the process will repeat itself.

Cadweld's tool management system is based on the so-called CAD Tool Knowledge Object (CTKO). CTKOs are created by linking a representation of the corresponding tool knowledge to a CAD tool. A CTKO consists of two basic components: The frame body and the control body. The frame body is collection of information which represents tool

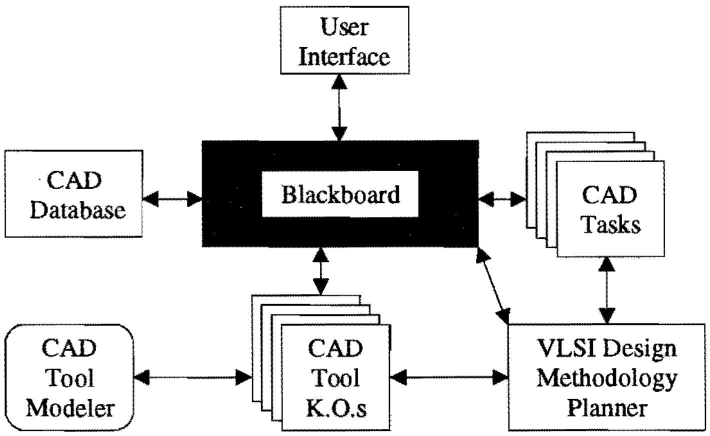


Figure 2.7: Cadweld's overall architecture.

invocation and tool characterisation related information. The tool characterisation part is used by the designer during tool selection. The *control body* consists of a set of activation patterns to which the tool can respond. It is used to control the activation of the CTKO in response to the state of the blackboard.

Cadweld CTKOs are organised by using a "CAD tool" class hierarchy like the one shown in Figure 2.8. Information in common to a certain "class" of CTKOs is concentrated in the corresponding class definition. "CAD Tool" is the most general class of the hierarchy presented in Figure 2.8. It describes tool invocation information which is common to all the tools. From the general CAD Tool class a number of subclasses are derived, which represent information in common to simulation, verification, inspection and design capture tools, respectively. From these classes even more specialised classes are derived, e.g., the "Simulators" class features three subclasses "Circ Sim", "Switch Sim", and "Timing Sim". CTKOs inherit information from the classes they belong to, e.g., the CTKO of a timing simulator is characterised by the information it inherits from the "Timing Sim", "Simulators" and "CAD Tool" classes. This prevents unnecessary duplication of this information and provides us with a mechanism to keep this information consistent for all the CTKOs belonging to a class. Further down the class hierarchy, the information becomes more specific. Control related information is usually found at the lowest levels.

Design flow management was added to Cadweld by the introduction of the so-called CAD Tasks. A CAD Task is a script, which describes a sequence of design steps as presented above. Because of the blackboard mechanism, these descriptions will not have to contain explicit information about the available CTKOs. Therefore, tools can be freely added or removed from the system without having any impact on the validity of the CAD Task

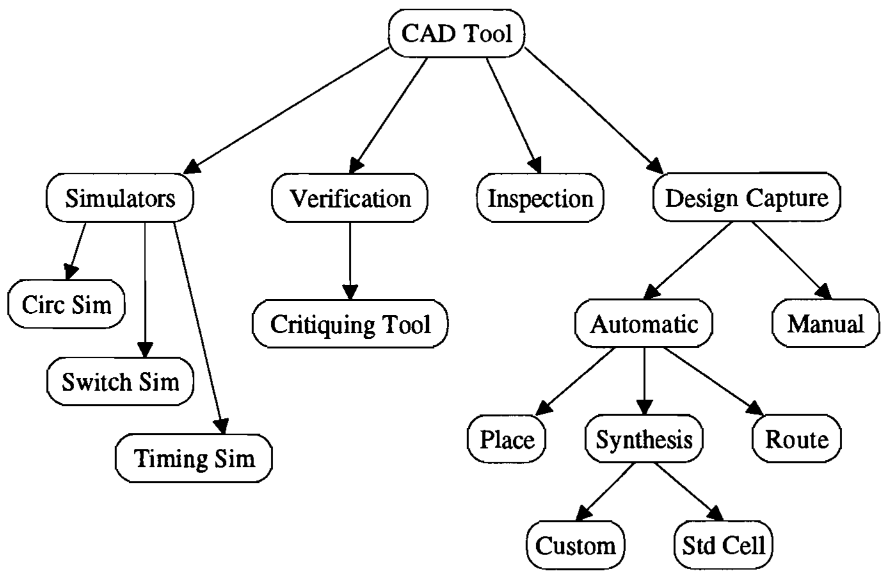


Figure 2.8: A CAD Tool class hierarchy.

descriptions. Like the basic tools, CAD Tasks can volunteer to perform a certain assignment posted on the blackboard, making them a kind of higher level tools.

2.5 The architecture of CAD frameworks

There is still much confusion about the general architecture of CAD frameworks. In this section we discuss two architecture proposals. The first proposal is treated because it gives a very detailed picture of a possible CAD framework structure. The second, much less detailed proposal, is considered because it has been widely accepted.

In [HNSB90] the very detailed structure of Figure 2.9 is proposed. According to this proposal a well designed CAD framework should have many layers of abstraction, of which the corresponding services are provided to the CAD framework users.

At the lowest abstraction layer we find the "Operating System", which provides facilities for manipulation and organisation of files (File Services), execution of programs (Process Services), communication via networks (Network Services) and interaction with the users of the system (User I/O Services).

The second layer is the "Abstract Operating System" view. This layer is required because not all operating systems provide the same services. The single abstract operating system

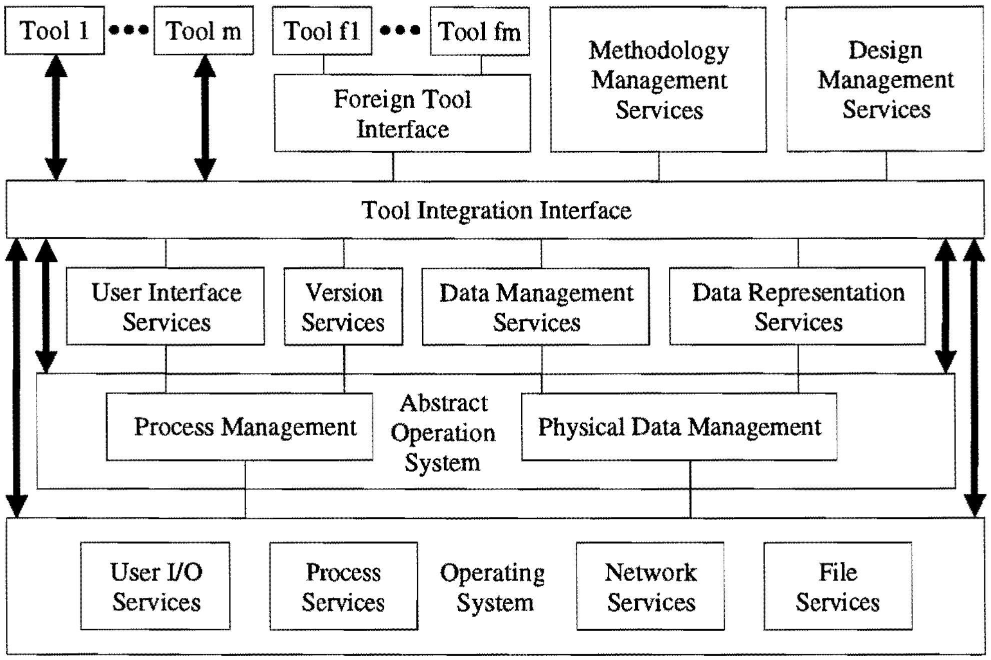


Figure 2.9: Components of a modern engineering framework.

eliminates these differences, thereby making the rest of the framework code independent of the specific operating system used. The services provided by this layer are "Physical Data Management" and "Process Management". These services enable the framework users to access raw data and to execute (CAD) programs without the need to know where these are stored on the network.

The third layer provides facilities for building user interfaces (User Interface Services), managing the CAD data associated with designs and co-ordinating access to these data by multiple designers or CAD tools (Data Management Services), managing the history of a design (Version Services), and facilities for describing what data items represent by giving the type of and the relations between these pieces of data (Data Representation Services).

The third layer enables the framework's users to retrieve the desired data and to activate CAD programs. However, these CAD programs have to be supplied with the data to operate on. As long as they are of the correct type, the tool arguments may be arbitrarily chosen. However, for most tools a fixed method is used to select the data they are applied to. By combining information about tool activation and the data selection method more abstract tools can be created. Data selection methods can be described in terms of interactions with the third layer. The tool integration interface provides a language to describe these interactions. When the source code of a tool is available it can be integrated directly into the framework by replacing its normal data access procedures by data selection methods. For many tools however the source code is not available. Such tools can be integrated into the framework by encapsulating them in such a way that the tool sees the input and output file formats it expects, while the data is actually being handled by the framework. The software which implements that encapsulation is referred to as the "Foreign Tool Interface".

Using the abstract tools as building blocks, Methodology Management Services can be provided. These services enable the system integrator or the end user to specify recipes, which describe in which order (sequentially or concurrently) tools have to be activated to create the desired design.

A more general but less detailed proposal is presented in Figure 2.10. This architecture has been widely accepted by standardisation bodies [LJ92] [CFI92] and by industry [Dig91]. This proposal explicitly distinguishes between "Domain Neutral Data" and "Domain Specific Data". The former are the meta data and the framework data the framework uses to provide its services. The latter are the raw design and tool data. The "Framework Kernel" features framework services like those provided by the lowest four layers of the previous proposal (up to and including the Tool Integration Interface) and the higher level services, like Methodology Management and Design Management, derived from these. Besides the integrated and the encapsulated design tools special "Framework Tools" like meta data browsers are distinguished.

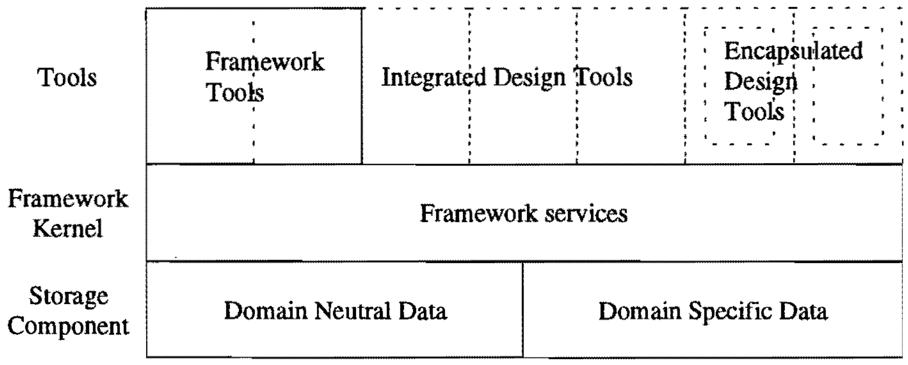


Figure 2.10: A general CAD framework architecture.

As an example of a real CAD framework complying with this general architecture, the structure of the Nelsis framework is presented in Figure 2.11 [vWBD90] [BvW90]. The storage component and tools depicted in Figure 2.11 can directly be mapped on those of Figure 2.10. Nelsis' framework kernel consists of a number of components which provide the framework services and a tool interface, referred to as the "Data Management Interface", which enables tools to use these services.

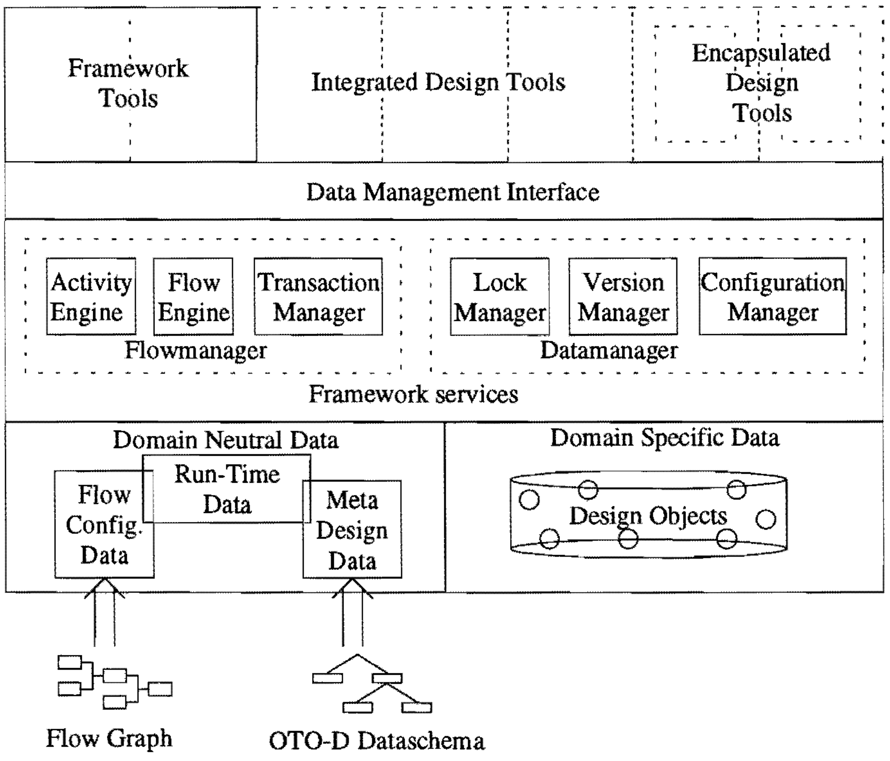


Figure 2.11: The Nelsis IC Design System architecture.

2.6 Summary

In this chapter we have presented the current ideas about what a CAD framework is and about what it is supposed to do, we have introduced the CAD framework participants and the current ideas about the three design management activities. This was followed by the presentation of two of the most complete existing CAD frameworks. The chapter was concluded by a discussion about the architecture of CAD frameworks.

Most design data management approaches are based on the ideas presented by Katz. As an example of how these ideas can be implemented in a real framework, the design data management system used for the Nelsis CAD framework was treated. Tool management was illustrated using the object oriented approach employed by the Cadweld CAD framework. Design flow management was demonstrated using the data flow and blackboard approaches employed by Nelsis and Cadweld, respectively. In the last part of this chapter two different proposals for the general structure of a CAD framework were presented, including a demonstration of how the structure of Nelsis can be mapped onto one of these general structures.

The general conclusion to be drawn from the literature regarding CAD frameworks is that there is a lot of confusion regarding the terminology employed, about what a CAD framework is, about who the participants are and about what the general architecture of a CAD framework should be. This despite the fact that some overview articles appeared trying to clarify these issues [HNSB90] [LJ92] [KGMB94]. Additionally, literature demonstrated that most of the existing CAD frameworks are not complete. This is illustrated by the fact that we had to use examples taken from different frameworks to introduce the design management and CAD framework concepts. Although providing support to all participants is one of the major characteristics of a complete CAD frameworks, this aspect receives relatively little attention in the existing literature and CAD frameworks. Another conclusion which can be drawn from literature is that there are lots of similarities between the different approaches. However, these similarities are clouded by differences in the employed terminology and by the fact that most papers mix implementation details with the underlying fundamental ideas. In this thesis we use the ideas these approaches have in common to create an abstract model of complete CAD frameworks. This model, which is presented in the coming four chapters, will enable us to reason about CAD frameworks without being bothered by implementation details.

Chapter 3

A CAD framework model

In this and the coming three chapters a CAD framework model is presented, which clearly demonstrates how a CAD framework can be constructed which not only provides support to all participants, i.e., to both the designers and DMS developers, but also features a good integration of design data, design tool and design flow management. As depicted in Figure 3.1, a CAD framework consists of two interacting components: A *Design Management System* (DMS) and a *DMS developer support system*. The design management system assists designers during their design management tasks. This system uses the data storage, data access and tool execution facilities of the operating system to provide its services. The DMS developer support system assists the DMS developers in their job of updating the design management system.

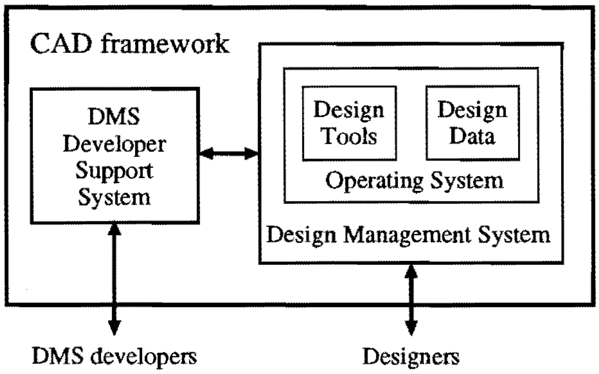


Figure 3.1: A CAD framework.

In Section 3.2 we demonstrate how a CAD framework can be constructed which provides assistance and a uniform interface to all participants. To achieve this, we use an analogy between designers and design management system developers. In this analogy all CAD framework participants are considered as designers who require the same kind of support.

The DMS of a complete CAD framework should not only provide support for design data, design tool and design flow management, but also has to feature a good integration of these three activities. In the last section of this chapter (Section 3.3) and the coming three chapters we present a DMS model which demonstrates how this integration can be achieved. This model is constructed by employing an analogy between digital systems and design management systems. It describes the design management system in terms of three interacting systems: A design data, a design tool, and a design flow management system.

We illustrate the DMS model by creating a description of a design management system automating the LOCAM timing-driven matching process. LOCAM, which is a logic synthesis system developed and distributed by Philips, will be used throughout this thesis to illustrate the concepts we introduce. In Section 3.1 a description of this synthesis system is given.

3.1 The LOCAM logic synthesis system

We will illustrate the concepts introduced in this thesis using LOCAM [Phi92]. LOCAM can be used to automatically synthesise a description of an IC in a High-level Design Language (HDL) into an optimised gate-level network. The general structure of LOCAM is depicted in Figure 3.3.

Currently the following three HDL input formats are supported:

- The Philips Hardware COmpiler language PHACO, a simple Pascal-like language,
- the Electronic Logic LAnguage ELLA [BGL92], a powerful and simulatable (UK-MoD) standard, and
- the VHSIC Hardware Description Language VHDL [VHD93], a powerful and extensive IEEE standard, simulatable on various levels.

Additionally, synthesis from netlists is supported via the Electronic Design Interchange Format (EDIF) [EDI93] and the hierarchical Netlist Description Language (NDL).

The heart of the LOCAM system is the Optimiser and MAtcher tool OMA. OMA is used to optimise Boolean expressions and to subsequently map these onto a standard cell library. The general structure of OMA is depicted in Figure 3.2.

OMA basically operates on designs represented using the Philips Logic And NETwork Specification format (PLANETS), which combines Boolean equations with structural descriptions. The different HDL and netlist descriptions are interfaced to OMA using the appropriate compiler to translate these representations to PLANETS. Additionally, OMA

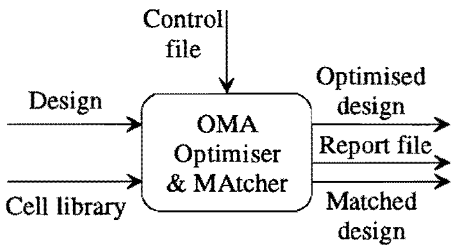


Figure 3.2: The Optimiser and MAtcher OMA.

can handle low-level input in the form of PLA tables and logic expressions. The operation of OMA is controlled by a control file. Among others, this file will denote the input file, determine how the design will be optimised, control whether or not matching will be performed, and if so, the library to be used, and finally what output will be generated. If matching is switched on, OMA requires an extra input, which is a library defining the standard cells onto which the logic expressions are to be mapped. The output of OMA consists of a file containing the optimised design (in PLANETS, PLA table or logic expressions format), a NDL netlist representation of the results of the matcher, and a report file listing all the messages produced during the OMA run. The NDL netlist produced by OMA can be converted using the OMA2<*language*> programs to the standard HDL and netlist formats or to the formats employed by design stations such as Mentor Graphics, et cetera.

3.2 The designer - DMS developer analogy

In order to construct a CAD framework which provides assistance and a uniform interface to all participants, we employ the following analogy between the activities of the two types of participants a CAD framework has to deal with: Designers and design management system developers. A CAD framework assists designers by managing their data, tools and flows. This task is performed by the corresponding design management system. The DMS is constructed and maintained by the DMS developers. Despite the differences between the tasks performed by all participants, they have one thing in common; they are all designers. The basic difference lies in what they design rather than how they design it. DMS developers design (parts of) a design management system. They do this by making or updating a description of (a part of) the design management system. These descriptions represent (a part of) the DMS at a certain level of abstraction and are expressed in an appropriate DMS description language. Like designers, DMS developers produce lots of data and use a variety of tools to do this. Therefore, DMS developers also require the CAD framework to assist them in managing their data, tools and flows. A CAD framework can

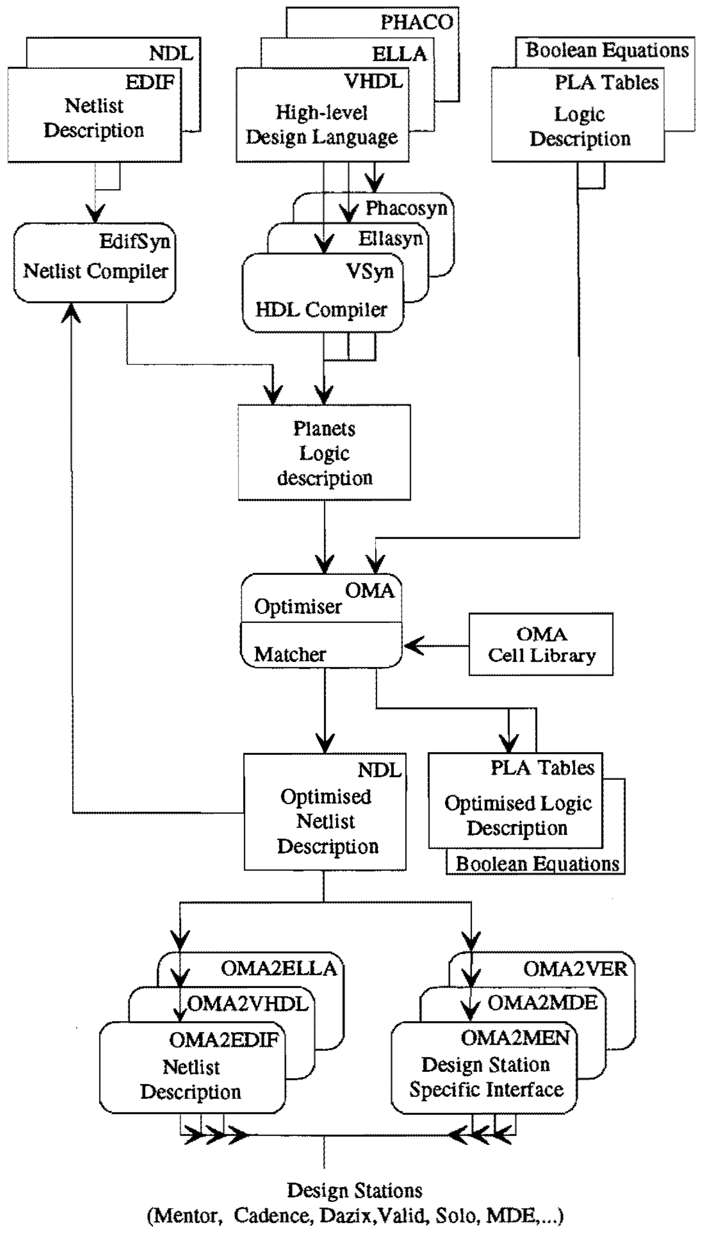


Figure 3.3: The LOCAM flow.

realise this by making the support facilities provided by its design management system available to the DMS developers.

Based on the designer - DMS developer analogy, we propose the CAD framework structure presented in Figure 3.4. In this proposal the design management system and the DMS developer support system of Figure 3.1 have been combined to form a more general design management system, which manages the data, tools and flows of both the designers and the DMS developers. Since all participants are supported by the same system, this automatically results in a uniform interface for everybody involved.

The way the DMS of Figure 3.4 manages the design process is not hard-coded into it. Instead, it consists of a DMS core which is independent from the design management strategy to be employed. This core is programmed by the DMS developers to perform its design management system services. Besides managing the data, tools and flows of the DMS developers, the DMS core also assists the DMS developers during its own programming process. It achieves this by allowing the DMS developers to program it by downloading DMS representations, rather than by modification of its source code. These representations describe a part of the DMS structure and/or behaviour at a certain level of abstraction and are expressed in one of the DMS description languages supported by the DMS core. In the rest of this thesis we present a detailed model of the DMS core and we show how this system can be programmed by the DMS developers to perform design management services.

3.3 The digital system analogy

Most CAD frameworks were constructed based on an informal and often incomplete model of the structure and behaviour of the corresponding design management system. The frameworks which use a formal approach are either based on a formal model of the design process [JD96] [SD96] or on a model of the design management activity [ASS95] [vW94]. Although these models are very useful because they lead to a better understanding about *what* a design management system is supposed to do, they give us little information about *how* this system should achieve this. Our approach features an explicit and mathematical model of the design management system. The model presented in this chapter highlights the relation between design data, design tool and design flow management. It is a generalisation of the models we presented previously [Rov90b], [Rov90a], [Rov94], which where based on the ideas presented in [Koo91] and which addressed design tool and design flow management rather than design data management.

Mehendale [Meh91] introduces a design flow management approach which is based on a parallel that can be established between the design flow and logic design domains. He uses this parallel to map design flows to logic level digital system representations (see Table 3.1), thereby enabling him to use the techniques and algorithms developed for the design of digital systems for the specification, representation and synthesis of design flows.

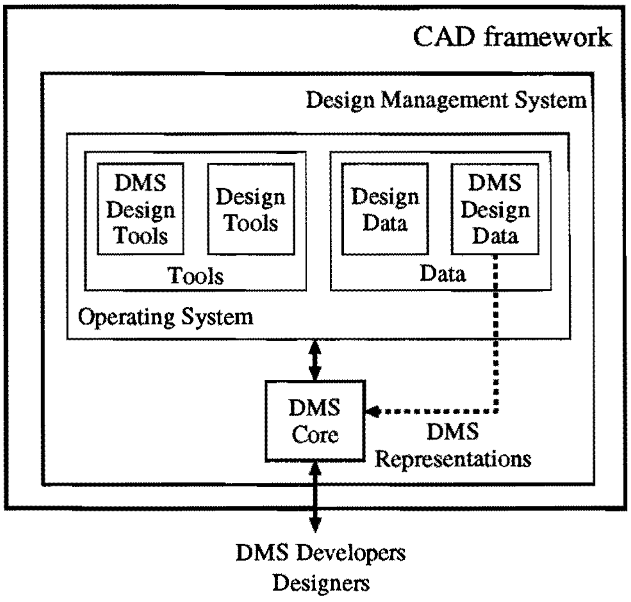


Figure 3.4: CAD framework structure based on the designer - DMS developer analogy.

Like Mehendale we make use of an analogy with digital systems. However, we use this analogy to model the complete design management system rather than for design flow representation. In Section 3.3.1 the general structure of digital systems is introduced. In Section 3.3.2 we use the digital system analogy to create a design management system model. This model describes the design management system in terms of three interacting components: A design data, a design tool, and a design flow management system. These three components, of which detailed models will be presented in the coming three chapters, are connected using busses. In Section 3.3.3 we show how these busses can be modelled.

3.3.1 Digital systems

The general structure of a digital system is presented in Figure 3.5 [Ein89].

Every digital system can be split into an *Information Processing Unit* (IPU), also referred to as a *datapath* or *operational unit*, and a *Control Unit* (CU), alternatively called *timing unit* or *controller.* Computations are performed by the IPU under control of the CU. The CU issues commands to the IPU using a number of *control signals.* It monitors the resulting operation of the IPU using the *status signals* supplied by the IPU.

Table 3.1: Design flow - logic design mappings.

design flow domain	logic design domain
tool	logic primitive
tool data	ports
data flow direction	port direction
program pathname	model
program options	model parameters
design flow	higher level cell
tool interconnects	nets
tool/subflow library	standard cell/mega modules library
repetitive flows	port arrays, cell arrays
iterative flows	sequential logic

The general structure of information processing units is depicted in Figure 3.6. IPUs consist of a data store, a data transformation unit and some data transportation channels connecting these.

Like its name suggests, the *data transformation unit* is used to perform transformations on data. It usually consists of a number of components, which we will refer to as data transformers. *Data transformers*, also referred to as *operators*, are the elements which execute the calculations performed by a digital system. Examples are combinatorial elements such as adders, multipliers and multiplexers. The operation of a data transformer is supervised by its control inputs, which determine what transformation it will perform at a certain moment in time. For example, consider the Arithmetic Logic Unit (ALU) data transformer of a microprocessor. An ALU can perform a number of operations on the data available at its data inputs (e.g. addition, subtraction, incrementation and decrementation). The control inputs of an ALU determine which of these transformations are actually being applied.

The *data store* consists of a number of storage elements (e.g. registers). Read and write access of these stores is supervised by the control signals. The data store supplies the data transformation unit with the required input data and stores the results for later use.

Data, control and status information is transported from and to the data transformation unit, the data store and the IPU environment by the *data transportation channels* connecting these. The operation of the data transformation unit and the data store is supervised using control signals. These IPU components in turn report their progress to their environment by producing status information. Hierarchy can be introduced in our digital system model by allowing data transformers and storage elements which are not primitive, but digital systems on their own, each characterised by their own CU and IPU.

The operation of the control unit can be represented using a *state transition graph*. A state transition graph is a labelled directed graph. The nodes and edges of such a graph can be

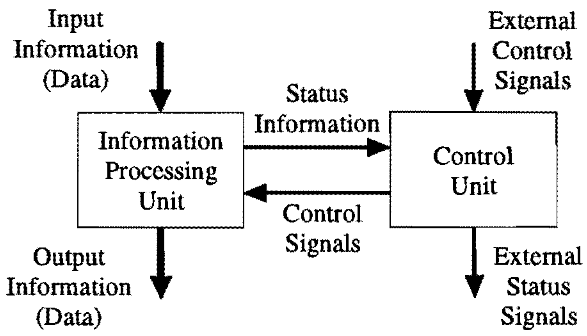


Figure 3.5: General structure of a digital system.

used to represent the states and transitions of the control unit, respectively. For the CU examples presented in this chapter, the labelling is done as follows. Every node is labelled with the output produced by the CU when it is in the corresponding state. Every edge is labelled with the CU input enabling the corresponding transition.

Digital systems are either synchronous or asynchronous. These systems differ in the way they change state. For a synchronous system the values of the IPU storage elements and the state of the CU change simultaneously. This synchronisation is achieved by making the occurrence of state transitions depend on a single signal, referred to as the *clock*. Asynchronous systems on the other hand, do not have such a centralised state transition mechanism, and therefore transitions will occur asynchronously.

As an example of a digital system, consider a small synchronous system which calculates the factorial of a natural number N ($N \geq 2$), denoted by N!, which is defined by

$$N! = N * (N-1) * (N-2) * \ldots * 2 * 1 \; .$$

A possible algorithm to calculate N-factorial is given by

$$\mathit{Factorial}(N) = \mathit{Fac}(N-1, N) \; ,$$

where

$$\mathit{Fac}(\mathit{Counter}, \mathit{Product}) = \begin{cases} \mathit{Product} & \text{if } \mathit{Counter} = 1 \\ \mathit{Fac}(\mathit{Counter} - 1, \mathit{Counter} * \mathit{Product}) & \text{otherwise} \end{cases} \; .$$

The structure of a synchronous digital system implementing this algorithm is represented in Figure 3.7. The data store of the factorial IPU consists of the registers *Counter* and *Product*, which are used to contain the values associated with the corresponding argument of the *Fac* function. The outputs of these registers are always enabled, so the stored values can be read at any time. Write access to the data store is supervised using the *hold* control signal, of which the inverted value is presented to the write enable inputs of the registers.

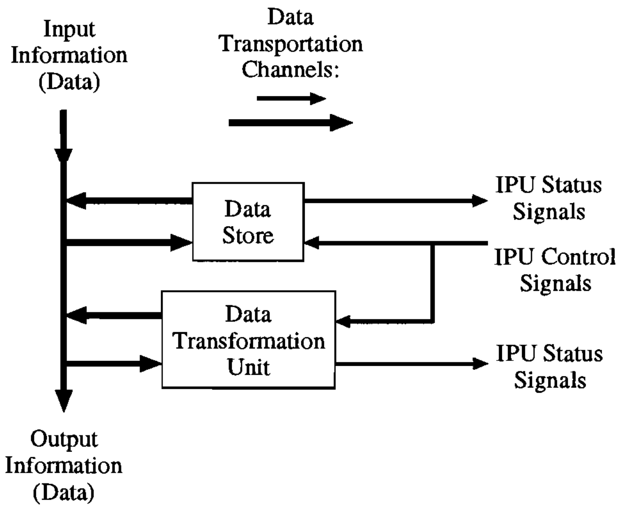


Figure 3.6: The general structure of an information processing unit.

The data transformation unit of our IPU contains three types of data transformers. The operators *Decr* and *Times* perform the actual calculations. They are used to decrement the value of *Counter* by one and to obtain the next value of *Product* by multiplying its old value with number stored in *Counter*, respectively. The multiplexers *MUX1* and *MUX2* select from which source the data is to be obtained from. The *One?* unit monitors the value presented to *Counter* and generates the status signal *one*, indicating whether this value is one or not. The components of the IPU are connected using data busses (thick arrows) and control and status lines (normal arrows). The arrowheads indicate the direction in which these data flow.

The factorial IPU is controlled by a control unit, which is a state machine with three states. In the start state *Ready* of this machine, the IPU performs no calculations. This is effectuated by the *hold* control signal, which disables the inputs of the *Counter* and *Product* registers. When the external control signal *start* is activated, the machine proceeds to the *Init* state. While the circuit is in this state, the *init* control signal will direct the multiplexers to select the external data input. This results in the initialisation of the registers *Counter* and *Product* to $N - 1$ and N, respectively. After the initialisation the *Calc* state is reached. In this state the recursive *Fac* function is calculated. The recursion is supposed to stop when *Counter* becomes equal to one. This is effectuated by the *one* status signal, which signals the CU that the calculation has finished. In response to the *one* status signal, the state machine will make a transition to the *Ready* state. In this state it will hold the IPU and raise the external status signal *ready*, indicating that the factorial

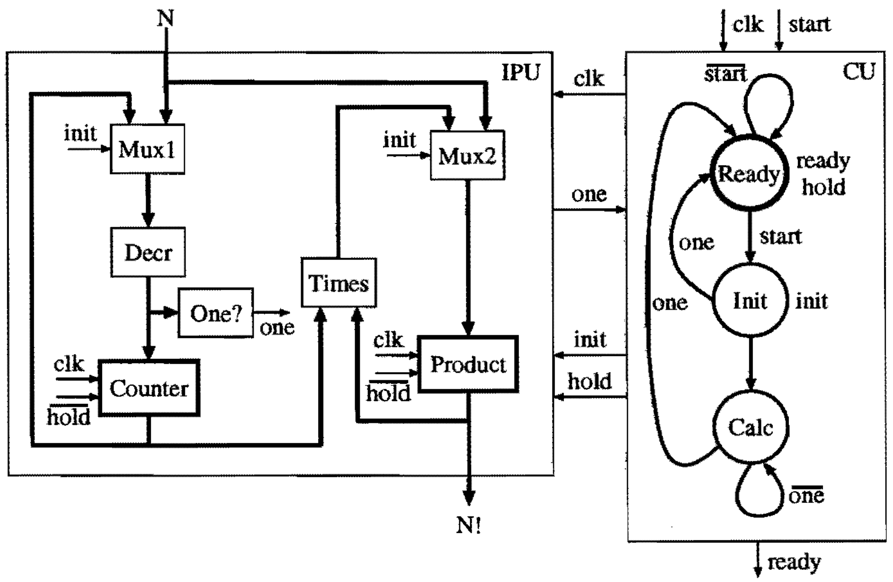


Figure 3.7: A synchronous digital system for factorial computation.

calculation has finished and that the value of $N!$ is available at the data output.

3.3.2 A design management system model

A *complete* design management system will in general consist of three co-operating subsystems, one for each design management activity. But how should these systems be combined? Making use of the digital system analogy, we arrive at the DMS structure presented in Figure 3.8. This figure shows a decomposed version of the DMS of Figure 3.4 consisting of three interacting subsystems: A design data management system, a design tool management system, and a design flow management system.

Like the data store of a digital system, the *design data management system* (DDMS) will store and control access to the data produced by the design management system. The design data management system consists of a number of design data storage elements (e.g. databases). The design and the integration into the DMS of these storage elements is the task of the *design data management system developers.*

The *design tool management system* (DTMS) is the equivalent of the data transformation unit in digital systems. The basic components of such a system, the tools, are designed by *CAD tool developers.* These tools are integrated into the DMS by *tool integrators.* The

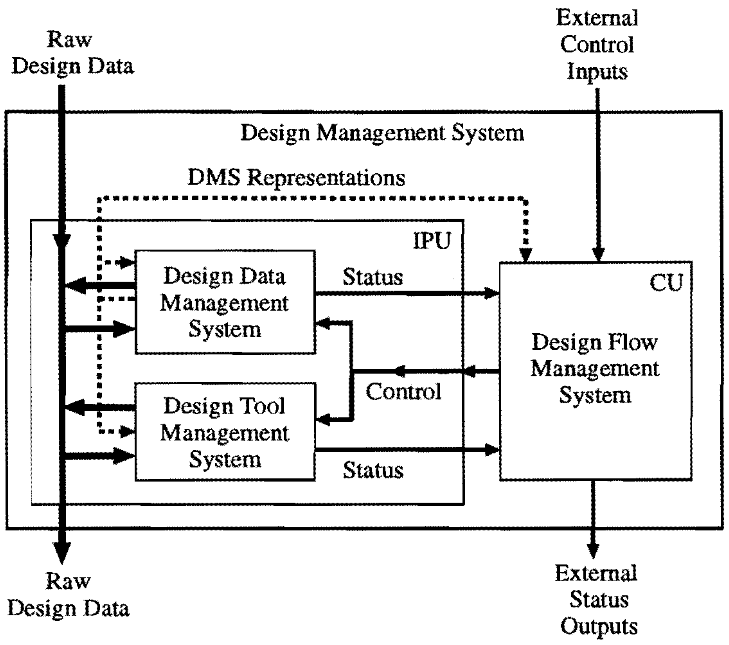


Figure 3.8: General structure of a design management system.

tool integrators hide tool invocation related information, thereby converting these tools into black boxes, which can be supervised using their control inputs and their status outputs. We also refer to these encapsulated CAD tools as *design data transformers.*

Like the digital system controller, the *design flow management system* (DFMS) can be represented by a state machine, which controls the operation of the DDMS and DTMS. This state machine is designed by the *design flow developers.* Based on the information it receives via the "Status" input, it guides the operation of the other two components by providing these with control input via its "Control" output.

Like the total design management system shown in Figure 3.4, its three components can also be programmed by the DMS developers. Therefore, in addition to the structure based on the digital system analogy, Figure 3.8 features some dashed lines indicating that the three design management components can be programmed by supplying these with DMS representations retrieved from the design data management system.

As an example of a DMS we present the asynchronous system depicted in Figure 3.9. This system supports the designer in using OMA to perform a form of timing driven matching, resulting in a good trade-off between area and timing. Designers can influence the OMA matching process by specifying timing constraints in the OMA control file. Our DMS example is based on a timing driven matching procedure described in [Phi92]. During the first step of this procedure the design is optimised (no matching performed). In the second step the optimised design is matched using overspecified timing constraints by requiring the maximum delay to be zero. This results in a circuit which is fast, but which occupies a large area. In the third step, the maximum delay of this circuit, obtained from the OMA report file, is used as a more realistic timing constraint for a new matching run. The resulting circuit will in general exhibit a good trade-off between area and timing.

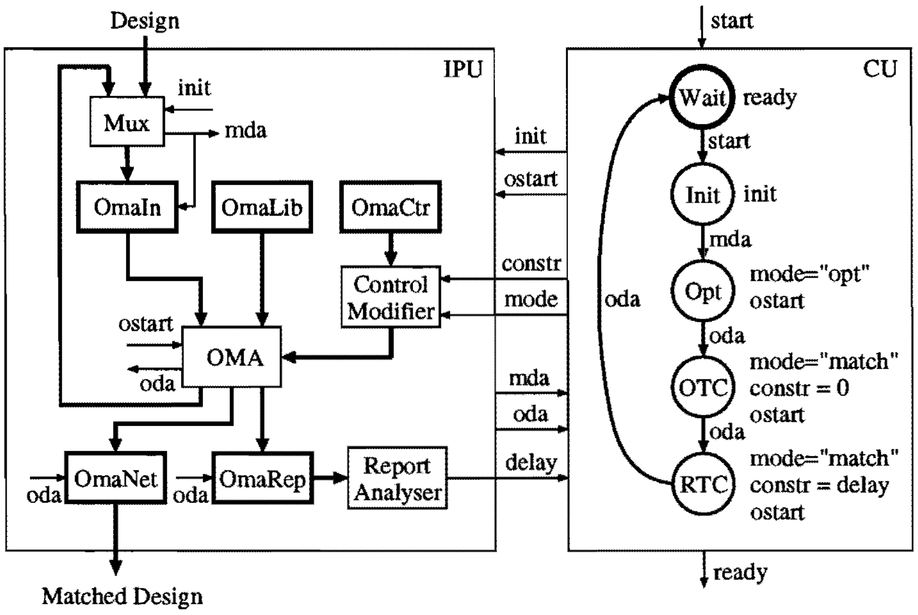


Figure 3.9: The OMA timing driven matching DMS.

The design data management system of the DMS of Figure 3.9 consists of a number of design data stores, one for each OMA input and output type. The outputs of these stores are always enabled, so their contents can be read at any time. Write access to these stores is controlled by a write enable input.

Like the data transformation unit of our factorial digital system, the tool management system contains three types of data transformers (tools), i.e., transformers which perform the actual calculations, transformers which select the source data is to be obtained from, and transformers which produce the IPU status signal by analysing the results of the calculations. The tools *OMA* and *Control Modifier* perform the actual calculations. They are used to optimise/match the input design and to fine tune the control file guiding this optimisation/matching process. *OMA* is activated by the OMA start signal *start* and reports its completion to its environment using the OMA data available signal *oda*. The behaviour of *Control Modifier* is controlled by its *mode* and *constr* inputs. The value associated with *mode* is either equal to "opt" or "match". This results in control files which put *OMA* in the optimisation or the matching mode, respectively. The timing constraint input *constr* is only meaningful in matching mode. The value of this input is a number which determines the timing constraint defined in the OMA control file. Like the multiplexers of the factorial system, the design data transformer *MUX* selects the source data is to be obtained from. When the selected design data become available at its output, the multiplexer signals this to its environment using the multiplexer data available signal *mda*. The *Report Analyser* is an example of a status generation tool. It is used to extract the maximum delay of a matched design from the corresponding report file.

The design flow management system of our timing driven matching system is represented by a state machine with five states. The machine is activated once it receives the external *start* command. In response it leaves its start-state *Wait* and goes to the initialisation state *Init*. In this state the control signal *init* is used to have the multiplexer select the externally presented design. In response to the *mda* signal this design is stored in *OmaIn* (the write enable input of this data store is connected to *mda*) and the state machine proceeds to the optimisation state *Opt*. In this state the design is optimised by *OMA*, which is achieved by setting *mode* to *"opt"* and by starting *OMA* using *ostart*. Once *OMA* is finished (denoted by *oda*), the machine proceeds to the Overspecified Timing Constraints *OTC* state. In this state the design is matched (*mode = "match"*) with a timing constraint of zero (*constr = 0*). The resulting *delay* of the matched design is used in the Realistic Timing Constraints *RTC* state as a more realistic timing constraint (*constr = delay*) for the matching process corresponding to this state. After this, the machine returns to the *Wait* state and raises the external status signal *ready* to indicate its completion.

3.3.3 Busses

As shown in Figure 3.8, the design flow management system controls the design data and design tool management systems by supplying these with control inputs via the control bus. It monitors the operation of these systems using the resulting status information received via the status bus. However, the situation sketched in this figure is a little bit too abstract. For example, if an active design flow sends a control input to the design data management system via the control bus, then how do the DDMS and DTMS know which one of them has to process it. Likewise, the design flow management system will not known whether the information available at the status bus was produced by the design data or the design tool management system.

There are two ways to solve the problem described above: Addressing or bus decomposition. In the first case address information is added to the messages send via the control, status and data busses. The DMS components connected to these busses then use this information to determine whether a message is meant for them or not. A disadvantage of addressing is that it results in an implicit representation of the communication structure. Another disadvantage is that it provides no protection; neither against messages being read by components they were not meant for, nor against control messages being produced by components which do not have the right to do this. In this thesis we will use an approach based on bus decomposition, which does not have these disadvantages. For an illustration of this approach, consider the design management system for the OMA Timing Driven Matching (OTDM) process depicted in Figure 3.9. For this DMS the control, status, and data busses depicted in Figure 3.8 have been decomposed into a number of smaller busses, which directly connect a data source to a data destination. The components of the OTDM DMS can only communicate with the components they are connected to via one of these busses. This not only results in an explicit representation of the communication structure of the OTDM DMS but in addition restricts communication to the components involved.

In contrast to the OTDM DMS, the design management system model of Figure 3.8 does not describe the design data and design tool management system in terms of a number of separate units, which can be connected using busses. Despite this, the problem described above can be solved by decomposing the design data, control and status busses of Figure 3.8 into a number of more specialised busses. Before we show how this can be done, we will first give a detailed description of what a bus is. Basically, busses are connections between a number of data sources and a number of data drains and are defined as follows.

Definition 3.3.1 *Bus*

A bus is a pair <*Cont, BusDef*>, of which *Cont* denotes the current contents and *BusDef* denotes the definition of the bus, respectively.

A bus definition is a 4-tuple <*Name, Sources, Drains, BusType*>, where

- *Name* - the bus identifier,
- *Sources* - a set containing the outputs via which data are put onto the bus,
- *Drains* - a set describing the inputs which get data from the bus, and
- *BusType* - the type of this bus.

A bus type is a 4-tuple <*ContType, RF, DF, WF*>, where

- *ContType* - the bus contents type,
- *RF* - the read function, which when invoked will read data from the bus,
- *DF* - the deletion function, which will result in the removal of data from the bus, and
- *WF* - the write function, which determines how new data are added to the bus.

The potential data sources in the DMS model of Figure 3.8 are the outputs of the three design management system components and the inputs of the DMS itself. Likewise, the data drains are the inputs of the three DMS components and the outputs of the DMS. From now on we will use the following names to refer to these sources and drains. The design data and design tool management system status outputs are referred to using "ddms.status" and "dtms.status", the corresponding design data outputs are named "ddms.data.out" and "dtms.data.out", the design flow management system control output "dfms.control", and the external control and design data DMS inputs "ext.control" and "ext.data.in". Likewise, we introduce the following names for the data drains: "ddms.control" and "dtms.control", "ddms.data.in" and "dtms.data.in", "dfms.status", and "ext.status" and "ext.data.out".

If a bus with a single source "dfms.control" and only one drain "ddms.control" is created, then it can be used by an active flow to send control inputs to the design data management system. The resulting status information can be delivered to the design flow management system using a bus with source "ddms.status" and drain "dfms.status. Busses like these, which only have a one source and drain, will also be referred to as *channels*.

The contents *Cont* of a bus is to a large extend determined by the values it currently transports. To illustrate this consider a bus which behaves like a One Place Buffer (OPB). Such a bus is only able to transport one value at a time. If a new value is written to an OPB bus, then the old value will be overwritten by the new one. The contents of an OPB bus is given by the value currently transported by it. Besides the data it currently transports, the contents *Cont* also depends on the way the bus organises these data. As an example, consider a bus which operates according to the First In First Out (FIFO) principle. A FIFO bus has to maintain the relative order of the data it currently transports. It can do this by grouping these data using a sequence, which represents this ordering. As an example consider a FIFO bus transporting naturals. If "1" was the first value put on this bus, followed by "2", "3", "4" and finally "5", then a possible way to represent this is by making the bus contents equal to the sequence $< 1, 2, 3, 4, 5 >$. If a value is retrieved from this bus, then the contents will change to $< 2, 3, 4, 5 >$. When subsequently the value "6" is written to it, the bus contents will become equal to $< 2, 3, 4, 5, 6 >$. In general, the contents *Cont* will not only represent what data are currently transported by the bus, but also how the bus organises these data. The type of *Cont* is determined by the bus contents type *ContType*, which are related as follows

$$Cont \in ContType \ . \qquad (3.1)$$

For an OPB bus transporting naturals the type is given by $\mathbb{N}$. The contents of our FIFO bus is given by a natural number sequence of arbitrary length, including the empty sequence, which is represented by a type equal to $\mathbb{N}^*$.

Data can be read from, deleted from and written to a bus using the read function *RF*, the deletion function *DF* and the write function *WF*, respectively. When applied to the bus' contents, the read function will return one of the values currently transported by the bus. For a OPB bus, this function will return its current contents. For a FIFO bus, the read function will return the "first in" value. For example, if the bus contents is given by $< 1, 2, 3, 4, 5 >$, then the read function will return the value "1". If a bus is not transporting data at the moment its contents is read, then the undefined value $\perp$ will be returned. Application of the read function will never alter the contents of a bus; it can only be used to inspect the contents. The deletion function *DF* however, will change the bus, namely by removing one of the transported values. In case of an OPB bus, application of the deletion function will change the bus' contents to $\perp$. For a FIFO bus with contents equal to $< 1, 2, 3, 4, 5 >$, application of the deletion function will change these contents to $< 2, 3, 4, 5 >$. New data are added to a bus using the write function *WF*. For OPB and FIFO busses, the write function will have one argument, which is the value to be added to this bus. If a value is written to an OPB bus, then it will become the new contents of the bus. When invoked for a FIFO bus with contents $< 2, 3, 4, 5 >$ using an argument equal to "6", the bus contents will be changed to $< 2, 3, 4, 5, 6 >$.

The read, delete and write functions of a bus are invoked by the components connected to it. A component is only allowed to read and/or delete elements from a bus if this bus is connected to one of its inputs. For example, the design flow management system of Figure 3.8 can only read from those busses which are connected to either its "dfms.status" or "ext.control" input. A component is only able to write to those busses which are connected to one of its outputs. For example, the design flow management system is only allowed to write to the busses connected to either its "dfms.control" or "ext.status" output.

For a bus *bus*, we will use *bus?* to refer to the value returned by the read function. Moreover, *bus-* will be utilised to denote the bus *bus* after invocation of the deletion function. In case the write function of a bus *bus* has only one argument *val*, then *bus!val* will be used to refer to this bus after *val* has been added to it by invocation of the write function.

3.4 Summary

In this chapter a CAD framework model was presented. This model clearly demonstrates how a CAD framework can be constructed which provides assistance and a uniform interface to all participants, i.e., to both the designers using it and the DMS developers maintaining it. To achieve this a CAD framework should feature a design management system to support the designers and a DMS developer support system to assist the DMS developers. For the CAD framework model presented in this chapter, we have made use of an analogy between designers and design management system developers, which considers all these CAD framework participants as designers requiring the same kind of support. Based on this designer - DMS developer analogy, we have proposed a CAD framework structure in which the design management system and the DMS developer support system have been combined to form a more general design management system, which manages the data, tools and flows of both the designers and the DMS developers. Since all participants are supported by the same system, this automatically results in a uniform interface for everybody involved.

The way the combined DMS manages design processes is not hard-coded into it. Instead, it consists of a DMS core which is independent from the design management strategy to be employed. This core is programmed by the DMS developers to perform its design management system services. Besides managing the data, tools and flows of the DMS developers, the DMS core also assists the DMS developers during its own programming process. It achieves this by allowing the DMS developers to program it by downloading DMS representations, rather than by modification of its source code. These representations describe a part of the DMS structure and/or behaviour at a certain level of abstraction and are expressed in one of the DMS description languages supported by the DMS core.

The design management system of a complete CAD framework should provide support for design data, design tool and design flow management. In this chapter we have introduced a DMS model which demonstrates how a CAD framework can be constructed which provides support for all three activities. This model is constructed by employing an analogy between digital systems and design management systems. It describes the design management system in terms of three interacting systems: A design data, a design tool, and a design flow management system. In the following three chapters we will develop more detailed models of the design data, the design tool and the design flow management systems occurring in this design management system model, respectively. The three DMS components communicate via busses. In the last part of this chapter we have presented a detailed model of these busses.

We have illustrated the DMS model by creating a description of a design management system automating the LOCAM timing-driven matching process. LOCAM, which is a logic synthesis system developed and distributed by Philips, was described in this chapter and will be used throughout this thesis to illustrate the concepts we introduce.

Chapter 4

The design data management system

In this chapter a detailed model is presented of the Design Data Management System (DDMS). Figure 4.1 shows how the DDMS is positioned within the design management system. The model presented in this chapter gives us guidelines of how to construct a design data management system which provides support and a uniform interface to both the designers and the design data management system developers.

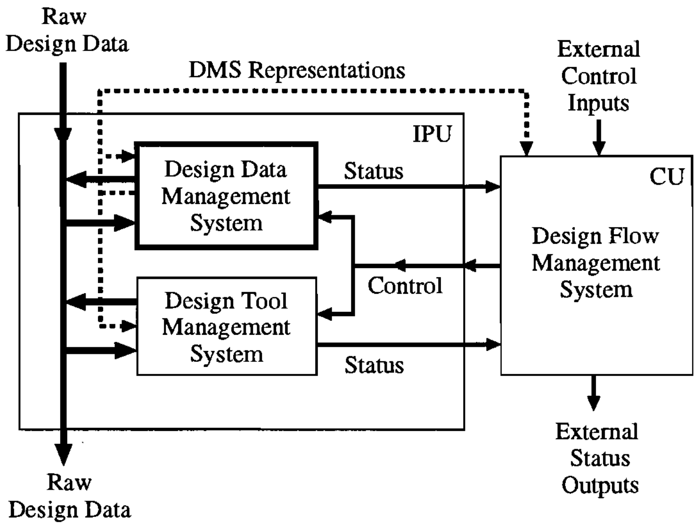


Figure 4.1: Environment of the design data management system.

Like the name suggests, a design data management system assists designers by managing their design data. Design data management can be considered as an activity involving

- design data storage,
- design data retrieval, and
- design data protection.

Design data storage and retrieval are strongly related activities. Data storage is useless if these data can not be retrieved afterwards. If a piece of design data is stored at a certain location, then this location has to be remembered until the moment these data are needed again. Typically a design data system will be concurrently accessed by multiple designers. This will lead to problems like designers trying to modify the same piece of data simultaneously. To avoid these problems most design data management systems will feature a design data protection mechanism.

The way a data store of a digital system organises its data is fixed, i.e., the available storage elements and their access mechanism do not change over time. This in contrast to the design data system. This DMS counterpart of the data store features a highly flexible design data organisation. For example, if a DDMS structures its design data space using workspaces, then this structure can be changed during its operation by addition or deletion of workspaces. It is the task of the design data management system developer to create and maintain a good DDMS organisation. A complete design data management system should support its developers in performing this task.

In Figure 4.2 the general structure of our DDMS model is depicted. It shows that the DDMS is modelled as consisting of two interacting components: The *design data store* (DDS) and the *control interpreter* (CI).

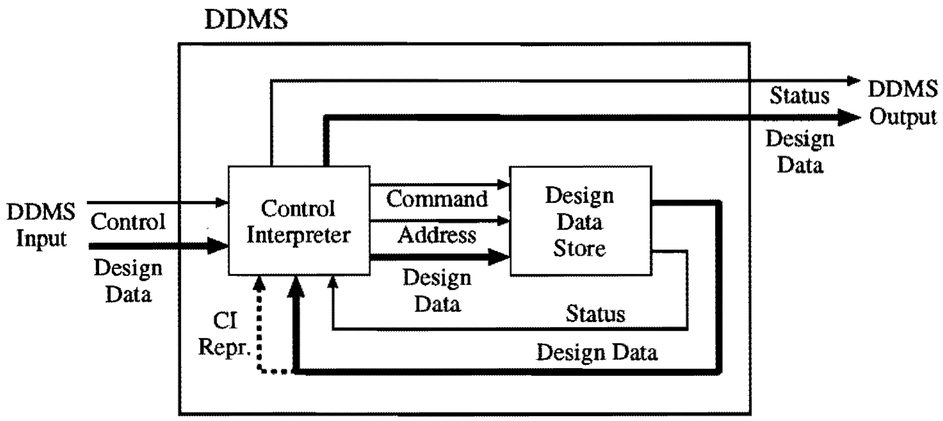


Figure 4.2: The design data management system model.

The design data store is the equivalent of the digital system data store. It will be used to model those aspects of design data management which are related to either design data storage, retrieval, protection or organisation. The design data store is a simple system featuring three inputs. These inputs are: The *command input*, which determines what action the DDS will perform (e.g. read, write or delete design data), the *address input*, which is used to (al)locate the data manipulated by such a command, and a *design data input*, via which the data to be stored are received. The DDS features two outputs, which present the status of the store and the design data read from the store to the store's environment.

The design data store provides a very simple interface to its environment. Each interaction consists of a single read, write or deletion operation. However, a design data management system will in general provide a more complex and higher level interface to its environment. We have modelled this by the introduction of a *control interpreter*. This DDMS component interprets the abstract control signals and the corresponding design data it receives from the DDMS's environment, by translating these into a sequence of simpler design data store inputs. It subsequently uses the resulting design data store output to generate the design data management system output, which consists of the design data retrieved from the DDMS and the DDMS status.

As indicated by the dashed arrow, the control interpreter can be programmed by the design data management system developers. The reason is that the answer to the question "Which abstract control signals should the control interpreter support?" is strongly determined by the design data management strategy to be employed. Therefore, the control interpreter will feature a language enabling the DDMS developers to specify which control signals it accepts and how these have to be translated into sequences of simpler design data store inputs.

In the first section of this chapter a detailed model of the design data store is presented. In Section 4.2 a detailed model of the control interpreter will be given. In the last section of this chapter, it will be demonstrated how the interaction of these two components results in the behaviour of the total design data management system.

4.1 The design data store

The operation of a design data store can be modelled by a state machine like the one depicted in Figure 4.3.

The data store of a digital system has a structure which does not change over time. Therefore, its state is completely determined by the data it currently stores. This in contrast to the design data store, which features a highly flexible design data organisation. For example, if a DDS structures its design data space using workspaces, then this structure can be changed during its operation by addition or deletion of workspaces. Therefore, the state of a design data store will not only be determined by the design representations it currently stores, but also by the way these data are currently organised.

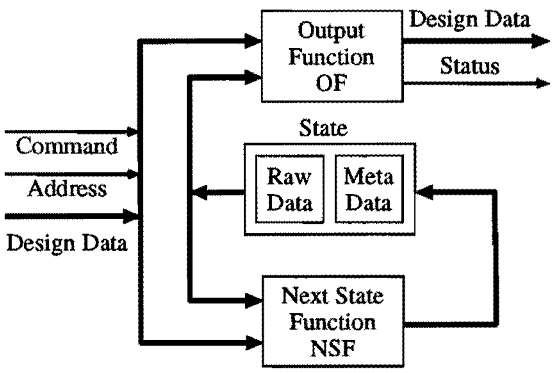


Figure 4.3: The design data store model.

Like depicted in Figure 4.3, the state of the DDS can characterised by two components: The *raw data set* and the *meta data set*. Raw data are the often large pieces of design data used and/or produced by designers and tools during the design process. The raw data set contains the raw data currently stored by the DDS. The meta data set on the other hand, contains information *about* the raw data managed by the store and information *about* how the store has currently organised these data.

For a design data store, the design data it stores and the status of its operation are made accessible to the store's environment by the *output function*. When issued a *read* command, the output function will retrieve the data specified by the corresponding address from the store's state. In this case the address will be a statement about the required properties of the data to be retrieved.

The state of a design data store is changed by its *next state function*. When issued a *write* command, the next state function will transform the store's state into a new state, containing the data presented at the store's design data input at the location specified by the value of the address input. *Delete* commands will result in data, located using the corresponding address, being deleted from the store's state by the next state function.

In our design data store model meta data is used to describe how the store is organised. Since these meta data are part of the DDS, they can be accessed by the design management system developers in the same way the other raw and meta data are accessed by the designers. So the design data store provides support and a uniform interface to both the designers and the design management system developers.

The output and next state functions of the design data store are independent from the design management strategy employed and can not be changed by the DMS developers. Therefore, these functions model a part of the DMS core. This in contrast to the store's state, whose contents is entirely determined by the data it stores and by the way these data have been organised by the DMS developers.

The rest of this section will be dedicated to the creation of more elaborate models of the different components of the design data store, which are: Raw data, meta data, the store's state, commands, addresses, the store's status, and finally the store's output and next state functions, respectively. Our treatment about meta data is split into three parts, each discussing a different type of meta data, namely raw data identifiers, attributes and relations. Raw data identifiers are labels used to uniquely identify every piece of raw data managed by the DDMS. Attributes are meta data used to represent the values of design data properties. Relations are meta data which are used to structure the DDS state by representing how the different state elements are related. We will show how these three types of meta data can be used to characterise raw data and to represent the store's organisation. Note however that the examples used to illustrate this are not part of our DDMS model and that the organisation presented by these examples is not meant as a description of the preferred way to structure the design data space.

4.1.1 Raw data

Raw data are the often large pieces of design data used and/or produced by designers and tools during the design process. The actual contents of the pieces of raw data differ greatly. Representations of a design at some level of abstraction expressed in a certain representation language, and auxiliary data, such as analysis results are all considered to be raw data. Raw data are distinguished from other design data by the fact that the actual contents of these data are of no importance for design management purposes; raw data are treated as black boxes by the design management system.

We will characterise a piece of raw data by specifying its type. Types can be represented by a set containing all the elements belonging to this type. For example, the GDS2-Layout type can be characterised by the following set

$$\text{GDS2-Layout} = \{\textit{layout} \mid \textit{layout} \text{ is a GDS2 layout}\}. \tag{4.1}$$

Usually such a set is not defined by enumeration of its elements, but by giving the syntax of the language in which these elements are expressed.

For most design data management purposes, it is sufficient to know what types exist and how these types are related, rather than how these types are defined. To illustrate this consider the types Layout, GDS2-Layout and L-Layout, defining layouts in general, layouts expressed in the GDS2, and layouts expressed in the L layout representation language, respectively. Although GDS2 and L descriptions are expressed in a different language, they are all layout representations. This fact can be represented by stating that

$$\text{GDS2-Layout} \subset \text{Layout, and} \tag{4.2}$$

$$\text{L-Layout} \subset \text{Layout.} \tag{4.3}$$

The types GDS2-Layout and L-Layout are said to be subtypes of the corresponding supertype Layout. Subtypes are usually created by defining a subset of an existing type. Supertypes are formed by unification of the corresponding subtype sets. For example the Layout set can be defined by

$$\text{Layout} = \text{GDS2-Layout} \cup \text{L-Layout}. \tag{4.4}$$

This not only states that GDS2-Layout and L-Layout are subtypes of Layout, but also that no other layout subtypes are considered.

Design descriptions are often hierarchical. A hierarchical design consist of a number of smaller designs. These design components are either primitive or composite. A primitive component is a complete description of one of subsystems the total system is composed of. A composite component on the other hand, describes a system in terms of a number of interconnected subsystems. Every subsystem used in such a component is not described there, but defined using a reference to one of the other (primitive or composite) design components. As an example consider the hierarchical microprocessor design depicted in Figure 1.3. At the top of this hierarchy we find the composite "microprocessor" design component, which describes how the controller and datapath subsystems interact. The primitive design component "controller" is a complete representation of the microprocessor's controller. The composite "datapath" component is described in terms of how the ALU, memory and multiplier primitives are interconnected.

Although it is possible to treat these hierarchical descriptions as a single piece of raw data, for design management purposes this is often impractical. To illustrate this consider a situation where a large design is changed a number of times, each change only affecting some of the components rather than the complete design. In this case the design data management system would have to maintain a number of very similar design versions, requiring an unnecessary amount of disk space. Another situation, in which it is inconvenient to treat a hierarchical design as a single unit, occurs when a design is created by a team of designers. In this case, the design data management system would either have to restrict access to this piece of data to one designer at a time or provide the designers with copies of the design and feature some complicated scheme to the merge the changes made to these copies.

A possible way to remedy these problems is to split up the hierarchical design description into a number of smaller but incomplete design descriptions. A natural way to do this is to partition the hierarchical design into groups of design components, each representing a part of the total design intended to be treated individually. This approach however, only works if the relations between these design parts are maintained.

We model the raw data storage of a design data store by the introduction of a raw data set *Raw*. Since raw data are treated as black boxes by the design management system, raw data set elements are distinguished using their physical storage location and type rather than their actual contents. For example, when a piece of raw data is copied to an other location, then this will result in two pieces of raw data with the same contents, but represented by different elements of the raw data set. Deletion of a piece of raw data is modelled by removing the corresponding element from the raw data set. If raw data is being overwritten by new data, then this is represented by replacement of the corresponding set element by a new element representing the new data. File move operations, e.g. using the UNIX command "mv", will in general only change the way a piece of raw data is referenced, i.e. they will not change the physical storage location of the raw data. Therefore, file move operations will in general have no effect on *Raw*.

As an example of a raw data set, consider the situation after the completion of the OMA timing driven matching process. When applied to a design named *Design* expressed in the input format *Format* and matched using a library named *Lib*, OMA assumes its design, control and library inputs to be stored in files named *Design*.*Format*, *Design*.oma_ctr and *Lib*.oma_lib, respectively. The optimised design, the netlist and the report file generated by OMA are stored in files named *Design*.oma_out, *Design*.oma_net and *Design*.oma_lis, respectively. The OMA timing driven matching process consists of three consecutive OMA runs, each overwriting the results of the previous run. We assume that no steps are taken to prevent this from happening. The result of the optimisation step *Design*.oma_out serves as the design input for the following two runs. This requires the optimised design to be moved to *Design*.*Format*. This will however overwrite the original input design, which is prevented by first moving this specification to *Design*.spec.*Format*. Under these circumstances the raw data situation after the timing driven matching run can be represented by a raw data set given by

$$Raw = \{Spec,\ Opt,\ Lib,\ Netr,\ Repr\}\ , \tag{4.5}$$

where *Spec* and *Opt* the original input design (specification) and the optimised design obtained from this specification during the optimisation step, *Lib* the library file, and *Netr* and *Repr* the netlist and report file produced during the realistic timing constraints matching step. The other raw data produced during the timing driven matching process do not occur in this set, because they were overwritten by the current elements of *Raw*. The three control files and the template used to generate these, are also omitted from the raw data set. The reason for this is that these files contain information about how to fine-tune the operation of OMA, rather than data related to the design at hand. Although it is possible to use the DDMS to manage these files, we will assume that these files are managed by the design tool management system instead.

4.1.2 Raw data identifiers

The raw data set models the raw data currently stored by the DDS. The DDS will however also contain information about data previously stored by the DDS, which do not belong to the raw data set anymore. In order to be able to represent information about raw data independent from their presence in the raw data set, we will assign a unique identifier to each piece of raw data managed by the DDMS. Information about a piece of raw data will be associated with this identifier rather than with the raw data itself. Because of this, raw data information can persist even if the corresponding raw data has been deleted. We represent the raw data identifiers currently employed by a DDS by the introduction of the raw data identifier set *RId*. In the rest of this section we introduce two different raw data identifier creation schemes. Note however that these are just examples of possible methods to generate raw data identifiers and not the only ones which can be devised.

Raw data are uniquely identified by their location in time and space. For example, if all the raw data managed by a DDMS are stored using a UNIX file system, then these raw data can be identified by specifying their path, filename and creation date, like "/user/rovers/micro.gds2" and "16:03-29/07/94". When these pieces of information are combined to form a raw data identifier, e.g. "/user/rovers/micro.gds2-16:03-29/07/94", and the raw data are never overwritten in the same minute they were created, then this identifier will be unique.

As an example of a raw data identifier set consider the following set, whose elements identify the raw data used and/or produced during the OMA timing driven matching process of a microprocessor design.

RId = (4.6)

{/user/rovers/lowpower.oma_lib-12:00-15/08/93,	A low power library
/user/rovers/micro.pln-16:03-29/07/94,	The original input design
/user/rovers/micro.oma_out-17:09-29/07/94,	The optimised design
/user/rovers/micro.oma_lis-17:09-29/07/94,	The report file of the Opt step
/user/rovers/micro.spec.pln-17:10-29/07/94,	The moved original input design
/user/rovers/micro.pln-17:10-29/07/94,	The moved optimised design
/user/rovers/micro.oma_net-17:18-29/07/94,	The result of the OTC step
/user/rovers/micro.oma_lis-17:18-29/07/94,	The report file of the OTC step
/user/rovers/micro.oma_net-17:27-29/07/94,	The result of the RTC step
/user/rovers/micro.oma_lis-17:27-29/07/94}	The report file of the RTC step

This example demonstrates a peculiarity of the employed identifier creation scheme, namely that a single piece of raw data can have a arbitrary number of raw data identifiers associated with it. For example the specification design is identified by both "/user/rovers/micro.pln-16:03-29/07/94" and "/user/rovers/micro.spec.pln-17:10-29/07/94". The reason for this is that file moves result in a new raw data identifier but do not create a new piece of raw data. Design data management of multiple identified raw data does not have to pose any problems. Information associated with raw data can be retrieved using the raw data identifier used at the moment this association was created.

Combination of path, file name and creation time is not the only way to create unique data identifiers. For instance, we could use the value associated with a global counter, which is incremented every time a piece of raw data is created, to identify such a newly created piece of data. In contrast to the previous one, this identifier creation scheme will associate exactly one identifier with each piece of raw data managed by the DDMS.

It does not matter which identifier creation method is employed, as long as it creates at least one unique identifier for each piece of raw data managed by the DDMS. In the rest of this chapter we will use the following more succinct identifiers for the raw data involved in the OMA timing driven matching process

$$RId = \{\text{lib, spec, opt, rep, neto, repo, netr, repr}\}, \quad (4.7)$$

which identify the library employed by OMA, the original input design, the optimised design, the report file of the Opt step, the netlist and report file produced during the OTC step and the netlist and report file of the RTC step, respectively.

4.1.3 Attributes

Attributes are meta data used to represent properties of either raw data or of the design data store's organisation. In this section we will give some typical examples of properties which can be represented using attributes.

As we have seen, raw data are uniquely identified by their location in space and time. This location can be characterised by attributes representing the values associated with the following properties:

- Medium - the medium the data are stored on,
- Position - the position of the raw data on this medium, and
- Time - the time the data was created.

IC design data are usually manipulated in computer memory, stored on a hard disk and archived on tape. The position on these media is determined by the address, track and block ranges these data occupy.

Although a piece of data is uniquely determined by the values of the "Medium", "Position" and "Time" properties, these are hardly of any use to a designer who wants to retrieve a specific piece of raw data. A possible way to remedy this problem is to additionally characterise a piece of raw data by making statements about the values associated with a number of more abstract properties such as

- Design - identifies the object (circuit, library, ...), during whose design process this piece of design data was created,
- Workspace - denotes the workspace in which this piece of data resides,
- Variant - identifies the design variant,
- Type - denotes the type of the design data,
- Version - specifies the design data's version,
- Creator - the process/person who created this piece of data, and

- Owner - the process/person who owns this piece of data.

For example, a GDS2 layout representation of a microprocessor μP to be used in a laptop, can be characterised by stating that the values associated with these properties are given by "μP", "private1", "portable", "GDS2-Layout", 1, "rovers", "rovers", respectively. Most operating systems enable their users to refer to the stored raw data using a label (e.g. path and filename) rather than the actual physical location. This label will in general be based on the values associated with properties like the ones presented above.

The organisation of the DDS is among others characterised by the workspaces employed, the relations which exist between these and their properties. Workspaces are characterised by attributes representing the values associated with properties such as

- WorkspaceName - the workspace identifier,
- WorkspaceType - the workspace type, which determines whether it is a project, group, private or archive workspace, and
- WorkspaceOwner - the owner of the workspace.

We model the attributes associated with a design data store by introduction of the set *Attr*, which contains all these data. In contrast to raw data, attributes are distinguished based on their contents rather than their physical storage location. So if a person's name is used in two different places, e.g. to identify both the owner of a piece of raw data and the owner of a workspace, this will give rise to only one *Attr* set element.

4.1.4 Relations

Data storage is useless if these data can not be retrieved afterwards. The data to be retrieved are located using a *characterisation* of these data in terms of the relations which exist between these and the other meta/raw data. Besides their usage for data characterisation, relations can also be used to represent how the design data store organises its data. This section will demonstrate how relations can be represented, typed and declared. In addition it will show how relations can be used for both data characterisation and representation of the DDS's organisation.

4.1.4.1 Relation representation

How are the relations which exist between meta and raw data represented? Relations group related elements, thereby distinguishing them from the unrelated elements. There are two ways to do this: Unordered or ordered. In the first case the related elements are combined by putting them in a unordered collection, e.g. a set. We will refer to this type of relation as a *grouping relation*. In the latter case, the elements to be related are ordered by grouping them using a sequence. A relation represented by a sequence will be referred to as a *sequencing relation*.

As an example of a relation represented by a set, consider the relation *redCardSymbols* represented by the set {hart, diamond}, which distinguishes the two symbols used on red coloured cards from the other two black coloured card symbols "club" and "spade".

The relation *bin6*, represented by the sequence $< 1, 1, 0 >$, which sequences the bits of the binary representation of the number 6, is a good example of a sequencing relation. For a sequence T, T_i can be used to refer to the i-th element of this sequence. For example the third element of the *bin6* sequence (0), is referred to as $bin6_3$. In the remainder of this chapter, we will use the term n-sequence when referring to a sequence of length n. So the sequence of the *bin6* relation is a 3-sequence.

For both grouping and sequencing relations we have a degenerated case, namely the relations represented by the empty set $\emptyset$ and the empty sequence ϵ, respectively. These special relations group zero elements.

4.1.4.2 Relation types

Relations are declared by stating to which relation type they belong. Like raw data types, *relation types* are represented by a set grouping a number of related relations. The *redCardSymbols* relation represented by the set {hart, diamond} groups the symbols "hart" and "diamond", thereby distinguishing these from the other symbols of the card symbol set {hart, diamond, club, spade}. So *redCardSymbols* defines a subset of the card symbol set. To reflect this, consider the following type for the *redCardSymbols* relation

$$\begin{aligned}&\{\emptyset, \{\text{hart}\}, \{\text{diamond}\}, \{\text{club}\}, \{\text{spade}\},\\ &\ \{\text{hart, diamond}\}, \{\text{hart, club}\}, \{\text{hart, spade}\},\\ &\ \{\text{diamond, club}\}, \{\text{diamond, spade}\}, \{\text{club, spade}\},\\ &\ \{\text{hart, diamond, club}\}, \{\text{hart, diamond, spade}\},\\ &\ \{\text{hart, club, spade}\}, \{\text{diamond, club, spade}\},\\ &\ \{\text{hart, diamond, club, spade}\}\}.\end{aligned} \tag{4.8}$$

This set contains all possible subsets that can be obtained from the card symbol set {hart, diamond, club, spade}. The type described above is a typical example of a kind of grouping relation type we will frequently employ, namely a set that is defined to be the power set of another set. The *power set* of a set A, denoted by $\mathcal{P}(A)$, is the set of all possible subsets of A (including the empty set and A itself). The type set of *redCardSymbols* can now be represented by

$$\mathcal{P}(\{\text{hart, diamond, club, spade}\}). \tag{4.9}$$

The *redCardSymbols* relation is declared by stating that

$$redCardSymbols \in \mathcal{P}(\{\text{hart, diamond, club, spade}\}). \tag{4.10}$$

As an example of a sequencing relation type consider the following possible type set for the *bin6* relation, given by

$$\{<0,0,0>,<0,0,1>,<0,1,0>,<0,1,1>,$$
$$<1,0,0>,<1,0,1>,<1,1,0>,<1,1,1>\}. \tag{4.11}$$

This set contains 3-sequences representing all possible 3-bit binary numbers.

Sequencing relation types can not be expressed as a power set of another set, because the elements of the type set are sequences rather than sets. There is however, a very similar concept for sequencing relation types. To illustrate this again consider the type set of the *bin6* relation. This set is equal to the following Cartesian product

$$\{0,1\} \times \{0,1\} \times \{0,1\}. \tag{4.12}$$

We will frequently encounter sequencing relation types which are defined by the Cartesian product of a number of other sets $A_1, A_2, \ldots, A_n$, denoted by $A_1 \times A_2 \ldots A_n$, and representing the set of all possible sequences $< a_1, a_2, \ldots, a_n >$, such that $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$.

We have demonstrated how we can succinctly represent the set of all 3 bit binary numbers. But what about the set of all binary numbers of arbitrary length? The corresponding type set is given by

$$\{0,1\} \cup (\{0,1\} \times \{0,1\}) \cup (\{0,1\} \times \{0,1\} \times \{0,1\}) \cup \ldots. \tag{4.13}$$

An alternative and finite notation for this set is $\{0,1\}^+$. In general, we will use A^+ to refer to the set $A \cup (A \times A) \cup (A \times A \times A) \cup \ldots$.

Since a type is represented by a set grouping the elements belonging to this type, thereby distinguishing them from elements not belonging to this type, a type can be considered as being a relation. A relation type is in fact a relation relating other relations. Therefore, we will also refer to a relation type as a *meta relation.*

4.1.4.3 Relation definitions

Until now we have encountered relation representations and relation types. A relation definition associates a relation representation with its type and with an identifier which can be used to refer to it.

Definition 4.1.1 *Relation definition*

A relation definition is a 3-sequence

<RelId, RelTypeDef, RelRepr>,

where

- *RelId* - the relation identifier,
- *RelTypeDef* - the relation type definition, and
- *RelRepr* - the relation representation.

As an example consider the following definition of the *redCardSymbols* relation.

$$<\text{redCardSymbols}, \mathcal{P}(\{\text{hart, diamond, club, spade}\}), \{\text{hart, diamond}\}>. \tag{4.14}$$

The relation type definition allows us to define a dynamic type, i.e., a type of which the set defining it changes over time. The reason why we need dynamic types is that we will frequently use relations of which the type is defined in terms of other sets. For example, we will often use relation types constructed from the *Raw*, *RId* and *Attr* sets, which represent the raw data, raw data identifiers and attributes currently managed by the design data store, respectively. Since the contents of these sets change over time, types referring to these will to. To reflect this dependency, a relation type is dynamically defined by the result of the evaluation of the corresponding relation type definition at the moment it is needed.

4.1.4.4 Relations and data characterisation

In this section we present examples of how relations can be used for data characterisation. Note however that the characterisation method presented here is not part of our design data store model and that it is only used to illustrate how relations can be used for this purpose. In addition, we do not claim to present a method to be preferred over those presented in [KBC+87] [vWSBD90] [vW93] [vW94].

Data are characterised by the way they are related to other meta/raw data. In order to be able to represent relations involving raw data independent from the presence of these data in the raw data set, we will use the corresponding identifier in these relations rather than the raw data itself. A necessary exception to this rule is the *raw data identification relation* "RIdRawMap", which we introduce to map raw data identifiers to the corresponding raw data. As an example consider the raw data identification relation representing the situation after the OMA timing driven matching process. It relates the elements of the raw data set represented by Equation 4.5 to the corresponding element of the identifier set of Equation 4.7 and is given by

$$\begin{aligned}&<\text{RIdRawMap}, \mathcal{P}(\mathit{RId} \times \mathit{Raw}),\\ &\quad \{<\text{spec}, \mathit{Spec}>, <\text{opt}, \mathit{Opt}>, <\text{lib}, \mathit{Lib}>, <\text{netr}, \mathit{Netr}>, <\text{repr}, \mathit{Repr}>\}>.\end{aligned} \tag{4.15}$$

There are two ways to obtain the raw data corresponding to a raw data identifier. Firstly, if this piece of raw data is an element of the raw data set, then it can retrieved using the raw data identification relation. Secondly, if the raw datum is not available anymore, but the data it was derived from are still stored in the DDS and the information about how this derivation was done is still known by the design methodology management system, then the raw data can be re-computed in close co-operation with the design tool and the design methodology management systems.

If the employed raw data identifiers are meaningful, raw data location can be based on this information. However, raw data identifiers are devised to be unique rather than meaningful.

Therefore, they will often be meaningless, e.g. the integers used in the global counter method, or based on low level information, e.g. when the medium, position on this medium and the creation time are combined to create a unique identifier. We do not want raw data identifier selection to be based on this implicitly represented low level information. We will remedy this problem by assuming the raw data identifiers to be meaningless and that selection is based on higher level meta data explicitly associated with these identifiers by means of relations. Meaningful identifiers are modelled by creating a meaningless identifier and by relating this identifier to meta data representing the identifier's original meaning.

One way to characterise raw data identifiers is by associating these with some higher level meta data, e.g. the corresponding design, workspace, variant, type, version, creator or owner. Such a characterisation can be used to locate raw data identifiers, but also to obtain the corresponding meta data once the identifier has been located. We model the link between raw data identifiers and attributes characterising these by the introduction of the *raw data identifier characterisation relation* "AttrRIdMap". This relation is represented by a set of pairs (2-sequences) <*key, rId*>, where *rId* a raw data identifier and *key* the attribute key characterising it.

As an example consider a simple raw data identifier characterisation relation corresponding to the situation occurring just after the optimisation step of the OMA timing driven matching process. The optimisation was performed by a designer "desl" in his own private workspace named "deslWS", using the "ALU" representation "spec" as the design input, resulting in an optimised version "opt" of this "ALU". As the key for this "AttrRIdMap" relation we have used 7-sequences <*design, workspace, variant, type, version, creator, owner*>. The corresponding relation is defined by

$$\begin{array}{l}<\text{AttrRIdMap},\\ \quad \mathcal{P}((\mathit{Design} \times \mathit{Workspace} \times \mathit{Variant} \times \mathit{Type} \times \mathbb{N} \times \mathit{Person} \times \mathit{Person}) \times \mathit{RId}),\\ \quad \{<<\text{ALU, deslWS, portable, OmaDes, 1, desl, desl}>, \text{spec}>,\\ \quad\ <<\text{library, libraryWS, low power, OmaLib, 3, libMan, libMan}>, \text{lib}>,\\ \quad\ <<\text{ALU, deslWS, portable, OmaDes, 2, desl, desl}>, \text{opt}>,\\ \quad\ <<\text{ALU, deslWS, portable, OmaLis, 1, desl, desl}>, \text{rep}>\}>.\end{array} \tag{4.16}$$

In the type definition of this relation we have assumed the existence of the sets *Design*, *Workspace*, *Variant*, *Type* and *Person*, which list the names of the design components, workspaces, variants, raw data types and persons currently involved in the design process, respectively.

The key of the gate library "lib" is very different from the rest, due to the fact that this piece of raw data is not specific for the design at hand. It denotes that for this OMA timing driven matching process, the "low power" variant of the OMA library is employed, which is available in the archive workspace "library" owned and created by library manager "libMan".

Raw data (identifiers) are not only located based on their attributes, but also using a variety of relations associating these to other raw data (identifiers). Typical examples are the "Version-of" and "Implements" relations. The "Version-of" relation represents that a piece of raw data is a modification/optimisation/version-of another piece of raw data. Both pieces of raw data will in general be described in the same language or format. The "Implements" relation combines a design at a certain level of abstraction to its specification at a higher level of abstraction. Both the "Version-of" and the "Implements" relation can be represented by a set of pairs. These pairs combine the identifiers of the two pieces of raw data involved, representing that the raw data identified by the first element of the pair is a "version of" and an "implementation of" the raw data associated with the second element, respectively. As an example consider the "Version-of" and "Implements" relations representing the situation after the OMA timing driven matching process, which are given by

$$<\text{Version-of}, \mathcal{P}(RId \times RId), \{<\text{opt}, \text{spec}>\}>, \text{ and} \tag{4.17}$$
$$<\text{Implements}, \mathcal{P}(RId \times RId), \{<\text{neto}, \text{opt}>, <\text{netr}, \text{opt}>\}>. \tag{4.18}$$

The "Version-of" relation relates the result of the OMA optimisation step, identified by *opt*, to the specification, identified by *spec*, it was derived from. The "Implements" relation combines the gate level result of the overspecified and realistic timing constraints matching steps, which are identified by "neto" and "netr", respectively, with their logical level specification identified by *opt*.

Besides their usage for data location, relations are also used to represent how the individually designed and/or managed components of a hierarchical design are combined to obtain a complete design. Although the different components of a hierarchical design can partly be designed and analysed separately, sometimes the complete design is required. This design can be obtained by combining the raw data of its components into a single piece of raw data. If a complete design is used as a single unit during the design process, then this will result in information being related to this complete design rather than to its individual components. This means that the complete design becomes one of the pieces of raw data managed by the system. The simplest way to do this is to store the complete design. This will however result in unnecessary data duplication, because the descriptions of its components are also stored by the system. An alternative approach is to store a representation of how the different components were combined to obtain the complete design, rather than the raw data itself. Such a representation, often referred as a *configuration*, contains the information necessary for the reconstruction of the complete design from its components. The configurations currently managed by a DDS are represented by the *configuration identification relation* "RIdConfigMap", which relates the descriptions of these configurations to the corresponding identifier. As an example consider the following relation describing two configurations "μP8" and "μP16", representing two microprocessor designs featuring an 8 bit and a 16 bit ALU, respectively.

$$\begin{array}{l}\langle\text{RIdConfigMap}, \mathcal{P}(RId \times \mathcal{P}(RId \times \mathcal{P}(RId))),\\ \quad\{\langle\mu\text{P8},\\ \qquad\{\langle\text{micro}, \{\text{dp8, control}\}\rangle,\\ \qquad\ \langle\text{dp8}, \{\text{ALU8, mem, mult}\}\rangle\}\rangle,\\ \quad\ \langle\mu\text{P16},\\ \qquad\{\langle\text{micro}, \{\text{dp16, control}\}\rangle,\\ \qquad\ \langle\text{dp16}, \{\text{ALU16, mem, mult}\}\rangle\}\rangle\}\rangle.\end{array} \tag{4.19}$$

Both microprocessors are divided in components according to the design hierarchy depicted in Figure 1.3. A configuration is represented by a set of pairs, each associating a design component with the subdesigns it uses in this configuration. For instance, in case of configuration "μP8", the pair <micro, {dp8, control}> relates the top level microprocessor design "micro" to the datapath and controller subsystem designs denoted by "dp8" and "control".

Like raw data, information associated with a configuration will be related to the corresponding raw data identifier, rather than to the configuration description itself. The only relations which can be used to determine if a raw data identifier refers to a piece of raw data or to a configuration are "RIdRawMap" and "RIdConfigMap". In all the other relations no distinction is made between normal raw data identifiers and configuration identifiers. As an example consider the situation in which the design corresponding to configuration "μP16" was obtained by optimisation of the design represented by configuration "μP8". In this case, "μP16" represents an optimised version of "μP8", which is reflected by the following "Version-of" relation

$$\langle\text{Version-of}, \mathcal{P}(RId \times RId), \{\langle\text{opt, spec}\rangle, \langle\mu\text{P16}, \mu\text{P8}\rangle\}\rangle. \tag{4.20}$$

Note that this relation makes no distinction between the configuration identifiers "μP8" and "μP16" and the normal raw data identifiers "spec" and "opt" used in our previous examples.

4.1.4.5 Relations and the DDS organisation

In the remaining part of this section we will demonstrate how relations can be used to represent the DDS's organisation. Again note that the examples presented here are not part of our design data store model and that these are only used to illustrate how relations can be used for this purpose. It is the task of the design data management system developer to create a useful DDS design data organisation. He will group data based on the design object, workspace, variant and/or type they belong to, and/or combine these based on their owner or creator.

The first thing about the DDS's organisation which will be represented is what the employed design objects, workspaces, variants and types are and which persons are involved. This information can be represented by relation definitions such as

$$
\begin{array}{ll}
\text{<Design, } \mathcal{P}(\mathit{Attr}) \text{, \{microprocessor, datapath, controller,} & \\
\qquad\qquad\qquad \text{ALU, memory, multiplier, library\}>,} & \\
\text{<Workspace, } \mathcal{P}(\mathit{Attr}) \text{, \{}\mu\text{PProjectWS, datapathWS, micconWS,} & \\
\qquad\qquad\qquad \text{des1WS, des2WS, des3WS, des4WS, libraryWS\}>,} & \\
\text{<Variant, } \mathcal{P}(\mathit{Attr}) \text{, \{desktop, laptop\}>,} & (4.21) \\
\text{<Type, } \mathcal{P}(\mathit{Attr}) \text{, \{OmaDes, OmaLib, OmaNet, OmaRep\}>, and} & \\
\text{<Person, } \mathcal{P}(\mathit{Attr}) \text{, \{des1, des2, des3, des4, libMan\}>.} &
\end{array}
$$

All these relations group a number of related attributes by defining a subset of the attributes set *Attr*. The relation identifier denotes what these attributes represent. The "Design" relation lists all the design object identifiers. In our example it contains the six individually designed components of the microprocessor hierarchy depicted in Figure 1.3 and the library used by OMA. The "Workspace" relation represents that the DDS data are organised using the eight workspaces represented in Figure 2.4. The "Variant" relation denotes that the design data store distinguishes two variants of the microprocessor design, namely a "desktop" and a "portable" low power variant. The "Type" relation describes which types of raw data the DDS is currently handling. "OmaDes", "OmaLib", "OmaNet" and "OmaRep", which are design, library, netlist and report data, respectively, are the types of the data used and/or produced by OMA. Note that these relations have been used previously in the description of the type of the "AttrRIdMap" relation of Definition 4.16. From now on we assume that for each set referred to in a relation definition other then *Raw*, *RId* and *Attr*, there exist a relation definition defining it.

The attribute subset definitions presented above give us information about what the basic units are of the DDS's organisation. These relations however provide no information about how these units are related. Design objects, workspaces, variants and types are usually organised in a hierarchy. Hierarchies can be represented by a relation, which relates the nodes of such a hierarchy to a set containing all the corresponding sub-nodes. As an example consider the following hierarchies:

$$
\begin{array}{ll}
\text{<DesignHier, } \mathcal{P}(\mathit{Design} \times \mathcal{P}(\mathit{Design})), & \\
\quad \text{\{<microprocessor, \{datapath, controller\}>,} & \\
\quad\; \text{<datapath, \{ALU, memory, multiplier\}>\}>,} & \\
& \\
\text{<WorkspaceHier, } \mathcal{P}(\mathit{Workspace} \times \mathcal{P}(\mathit{Workspace})), & \\
\quad \text{\{< }\mu\text{PProjectWS, \{datapathWS, micconWS\}>,} & \\
\quad\; \text{<datapathWS, \{des1WS, des2WS\}>,} & \\
\quad\; \text{<micconWS, \{des3WS, des4WS\}>,} & (4.22) \\
\quad\; \text{<library, \{des1WS, des2WS\}>\}>, and} & \\
& \\
\text{<VariantHier, } \mathcal{P}(\mathit{Variant} \times \mathcal{P}(\mathit{Variant})), & \\
\quad \text{\{<desktop, \{\}>,} & \\
\quad\; \text{<laptop, \{\}>\}>.} &
\end{array}
$$

The hierarchy relations "DesignHier" and "WorkspaceHier" represent the design and workspace hierarchies depicted in Figure 1.3 and Figure 2.4, respectively. The available variants "desktop" and "portable" are unrelated, which is represented by making these variants the top-node of their own empty hierarchy.

The relation definitions presented above represent what the basic units of the DDS's organisation are and how these are related. These definitions however present no information about how the design data are actually grouped using these units. To represent (part of) this information, we will introduce the "Designs", "Variants", "Types" and "Workspaces" relations, which represent how the design data are grouped according to the design object, variant, data type and workspace they belong to, respectively.

The "Designs" relation groups raw data according to the design object, during whose design process they were created. To illustrate this consider the following "Designs" relation, which organises the identifiers of the raw data produced during the OMA timing driven matching process of an ALU and the microprocessor parts used in the configurations of Equation 4.19 based on the corresponding design object

$$\begin{aligned}
&<\text{Designs}, \mathcal{P}(\mathit{Design} \times \mathcal{P}(\mathit{RId})),\\
&\quad \{<\text{microprocessor}, \{\text{micro}\}>,\\
&\quad\ \ <\text{controller}, \{\text{control}\}>,\\
&\quad\ \ <\text{datapath}, \{\text{dp8}, \text{dp16}\}>,\\
&\quad\ \ <\text{ALU}, \{\text{ALU8}, \text{ALU16}, \text{spec}, \text{opt}, \text{rep}, \text{neto}, \text{repo}, \text{netr}, \text{repr}\}>,\\
&\quad\ \ <\text{memory}, \{\text{mem}\}>,\\
&\quad\ \ <\text{multiplier}, \{\text{mult}\}>,\\
&\quad\ \ <\text{library}, \{\text{lib}\}>\}>.
\end{aligned} \tag{4.23}$$

If the microprocessor parts used in the configurations example were produced during the design of the "desktop" variant of a microprocessor while the timing driven matching process of the ALU was performed during the design process of the "laptop" variant of this microprocessor, then this can be represented by the following "Variants" relation

$$\begin{aligned}
&<\text{Variants}, \mathcal{P}(\mathit{Variant} \times \mathcal{P}(\mathit{RId})),\\
&\quad \{<\text{desktop}, \{\text{micro}, \text{control}, \text{dp8}, \text{dp16}, \text{ALU8}, \text{ALU16}, \text{mem}, \text{mult}\}>\ ,\\
&\quad\ \ <\text{laptop}, \{\text{lib}, \text{spec}, \text{opt}, \text{rep}, \text{neto}, \text{repo}, \text{netr}, \text{repr}\}>\}>.
\end{aligned} \tag{4.24}$$

The data produced during the OMA timing driven matching process can additionally be grouped according to their type, which is represented by the introduction of the "Types" relations, given by

$$\begin{aligned}
&<\text{Types}, \mathcal{P}(\mathit{Type} \times \mathcal{P}(\mathit{RId})),\\
&\quad \{<\text{OmaDes}, \{\text{spec}, \text{opt}\}>,\\
&\quad\ \ <\text{OmaLib}, \{\text{lib}\}>,\\
&\quad\ \ <\text{OmaNet}, \{\text{neto}, \text{netr}\}>,\\
&\quad\ \ <\text{OmaRep}, \{\text{rep}, \text{repo}, \text{repr}\}>\}>.
\end{aligned} \tag{4.25}$$

The "Design", "Variants" and "Types" relations presented above only group raw data identifiers but not the associated attributes. As an example of how both raw data identifiers and attributes can be grouped based on the workspace these belong to, consider the "Workspaces" relation, which is given by

$$
\begin{array}{l}
<\text{Workspaces}, \mathcal{P}(\mathit{Workspace} \times \mathit{WorkspaceType} \times \mathit{Person} \times \mathcal{P}(\mathit{RId} \times \mathit{WSStatus})), \\
\quad \{<\mu\text{PProjectWS, ProjectWS, des4,} \\
\qquad \{<\text{dp8, Ac}>, <\text{ALU8, Ac}>, <\text{mem, Ac}>, <\text{mult, Ac}>\}>, \\
\quad <\text{datapathWS, GroupWs, des2,} \\
\qquad \{<\text{dp16, Ca}>, <\text{ALU16, Ca}>\}>, \\
\quad <\text{micconWS, GroupWs, des4, } \{<\text{micro, Ac}>\}>, \\
\quad <\text{des1WS, PrivateWS, des1,} \\
\qquad \{<\text{dp8, Ap}>, <\text{ALU8, Ap}>, <\text{mem, Ap}>, <\text{mult, Ap}>\}>, \\
\quad <\text{des2WS, PrivateWS, des2, } \{<\text{dp16, Pro}>, <\text{ALU16, Pro}>\}>, \\
\quad <\text{des3WS, PrivateWS, des3, } \{<\text{control, New}>\}>, \\
\quad <\text{des4WS, PrivateWS, des4, } \{<\text{micro, Ap}>\}>\}>.
\end{array}
\tag{4.26}
$$

This relation not only represents to which workspace(s) a piece of raw data belongs, but also associates each workspace with its type and owner. The possible workspace types are represented by the following relation

$$<\text{WorkspaceType}, \mathcal{P}(\mathit{Attr}), \{\text{ProjectWS, GroupWs, PrivateWS, ArchiveWS}\}> . \tag{4.27}$$

Additionally, it associates each piece of raw data with its status in the workspace it occurs. We have used the workspace status values introduced in [RRvHK93], which are given by

$$<\text{WSStatus}, \mathcal{P}(\mathit{Attr}), \{\text{IPg, New, Pro, Ac, AcP, Ap, Ca, Iv, Old}\}>, \tag{4.28}$$

representing the following facts about the corresponding raw data: "construction In Progress", "New", "Promoted", "Accepted for children", "Accepted and Promoted", "Accepted by parent", "Candidate", "Invalid", and "Old", respectively. For more details we refer to [RRvHK93].

A possible interpretation of the "Workspaces" relation presented above is as follows. It represents a situation in which a microprocessor is designed by four designers whose work is co-ordinated using seven workspaces. In his private workspace "des1WS" designer "des1" has created an 8-bit datapath design "dp8" and the design components "ALU8", "mem" and "mult" used in this design. After completion these designs were promoted to and subsequently accepted by the parent workspace "datapathWS", which is indicated using workspace status "Accepted by parent" ("Ap"). The decision to accept these data was made by the owner "des2" of "datapathWS. "des2" in turn promoted these designs to the project workspace "μPProjectWS". After their acceptance by the project workspace owner "des4" (status "Ac" in this workspace), these designs were removed from "datapathWS". Designer "des2" has finished a faster 16-bit version of the datapath and the corresponding ALU ("dp16" and "ALU16") and promoted (status "Pro") these data to "datapathWS". These promoted data are candidates (status "Ca") for being accepted by "datapathWS".

In his private workspace designer "des3" has created the controller design "control", but has not yet promoted this design to the parent workspace "micconWS" (status "New"). Designer "des4" has created a version of the top level microprocessor description, which has been promoted to and accepted by "micconWS".

4.1.5 The design data store state

The design data store state (see Figure 4.3) describes which raw and meta data are currently stored by the DDS. The DDS state can be represented by a sequence combining all these data. We will represent the *design data store state* using the 4-sequence <*Raw, RId, Attr, Rel*>. Its first three elements consist of the raw data, raw data identifier and attributes sets we have already encountered. The last element is the relation set *Rel* which groups the definitions of the relations currently managed by the DDS.

Design data storage is only useful if the stored data can be retrieved at the moment they are required. In order to guarantee proper data retrieval, the design data store state should satisfy the following requirements.

1. *Every piece of raw data managed by a DDS needs to have a unique characterisation:* Raw data are characterised by the way they are related to other raw and meta data. The DDS uses these relations to retrieve these raw data and related information. If the characterisation of a piece of raw data is not unique, then it is not possible to distinguish it from the other data sharing this characterisation.

2. *Raw data have to be characterised correctly:* Although this requirement seems trivial, it is very important. Incomplete or incorrect characterisations can result in data not being found or even worse, wrong data being retrieved.

3. *Required data should be available:* If all raw data produced during the design process would be stored, then we would very soon run out of available disk space. Therefore, the design data management system will not keep all these data in its store, which is modelled by the absence of these data in *Raw*. This process however should only result in the elimination of obsolete data and/or data which can be re-created from other, still available, raw data.

Raw data are said to be *managed* by a DDS if and only if the corresponding raw data identifier belongs to *RId*. A DDS locates raw data and the associated information by employing its relations to obtain the corresponding raw data identifier. Once this identifier is known, it can be used to retrieve these data. Raw data identifiers are characterised by the way they are related to other data. If the raw data identifier characterisation is not unique, then it will be impossible to distinguish it from the other identifiers sharing this characterisation. So the first requirement can be rephrased as "Every *raw data identifier* needs to have a unique characterisation".

The first requirement is not satisfied by all DDS states. For example, consider a state featuring the following DDS relation

$$
\begin{aligned}
&\mathit{Rel} = \\
&\quad \{\text{<AttrRIdMap, } \mathcal{P}((\mathit{Design} \times \mathit{Type}) \times \mathit{RId}), \\
&\qquad \{\text{<<ALU, OmaDes>, spec>, <<ALU, OmaDes>, opt>,} \\
&\qquad\ \text{<<library, OmaLib>, lib>, <<ALU, OmaRep>, rep>}\}\text{>,} \\
&\quad\ \text{<RIdRawMap, } \mathcal{P}(\mathit{RId} \times \mathit{Raw}), \\
&\qquad \{\text{<spec, } \mathit{Spec}\text{>, <opt, } \mathit{Opt}\text{>, <lib, } \mathit{Lib}\text{>, <rep, } \mathit{Rep}\text{>}\}\text{>}\}.
\end{aligned}
\tag{4.29}
$$

This relation only consists of a raw data identifier characterisation relation and a raw data identification relation which represent the situation after the optimisation step of the OMA timing driven matching process. The keys of the characterisation relation are pairs <*design, type*> rather than the 7-sequences we introduced before. This represents a situation where the design object name and the type are the only attributes maintained during the design process.

For such a DDS state it is not possible to distinguish between the input design identifier "spec" and the optimised design identifier "opt". The reason for this is that both identifiers are characterised by the same key, namely <ALU, OmaDes>. Furthermore, the raw data identification relation only states that both "spec" and "opt" identify a piece of raw data currently stored by the system. This relation gives no further information which could be used to distinguish between these identifiers, because the related raw data *Spec* and *Opt* can not be distinguished themselves. Raw data are treated by the DDS as black boxes and can therefore only be distinguished based on their relation to other data. However, the raw data identification relation is the only relation in which *Spec* and *Opt* occur. As a result, the distinguishability of *Spec* and *Opt* will depend on the distinguishability of "spec" and "opt". Therefore, we conclude that "spec" and "opt" (and therefore also *Spec* and *Opt*) are indistinguishable.

The fact that a design data store state violates the first requirement can be corrected by providing some additional characterisation information. For instance, the DDS state of our example can be made to satisfy the first requirement by additionally monitoring the "Version-of" relation. Addition of this relation to the DDS relation will change its definition to

$$
\begin{aligned}
&\mathit{Rel} = \\
&\quad \{\text{<AttrRIdMap, } \mathcal{P}((\mathit{Design} \times \mathit{Type}) \times \mathit{RId}), \\
&\qquad \{\text{<<ALU, OmaDes>, spec>, <<ALU, OmaDes>, opt>,} \\
&\qquad\ \text{<<library, OmaLib>, lib>, <<ALU, OmaRep>, rep>}\}\text{>,} \\
&\quad\ \text{<Version-of, } \mathcal{P}(\mathit{OmaDes} \times \mathit{OmaDes}), \{\text{<opt, spec>}\}\text{>,} \\
&\quad\ \text{<RIdRawMap, } \mathcal{P}(\mathit{RId} \times \mathit{Raw}), \\
&\qquad \{\text{<spec, } \mathit{Spec}\text{>, <opt, } \mathit{Opt}\text{>, <lib, } \mathit{Lib}\text{>, <rep, } \mathit{Rep}\text{>}\}\text{>}\}.
\end{aligned}
\tag{4.30}
$$

Addition of the "Version-of" relation enables us to distinguish between "spec" and "opt", because it represents that "opt" was derived from "spec".

For checking raw data distinguishability we introduce the predicate *Distinguishable*. When applied to two raw data identifiers r_1 and r_2 and a DDS state *State*, *Distinguishable*(r_1,r_2,*State*) will determine if these two raw data identifiers can be distinguished in the context of this state. The formal definition of this predicate and a demonstration, of how this definition can be used to prove that $\neg$*Distinguishable*(spec,opt,*State*) and *Distinguishable*(spec,opt,*State*) hold for the states associated with the DDS relations of Equation 4.29 and Equation 4.30, respectively, is given in Appendix A.

Using the *Distinguishable* predicate, raw data identifier characterisation uniqueness can now be defined as

Definition 4.1.2 *Unique characterisation*

A raw data identifier $r_1 \in RId$ is said to be *uniquely characterised* by design data store state *State* if and only if for all other raw data identifiers $r_2 \in RId$ *Distinguishable*$(r_1, r_2, State)$ holds.

Unlike determining raw data characterisation uniqueness, it very difficult to see if the DDS state satisfies the second requirement "Raw data have to be characterised correctly." Incomplete or incorrect characterisations can to some extent be avoided by using relation types to ensure that the relations are consistent with their type. However, typing can not be used to avoid raw data being associated with the wrong data if these data are of the correct type.

When checking the third requirement "Required data should be available", the following two problems are encountered. Firstly, "Which raw data are *required* now or in the future?" A possible solution for this problem is to consider all the raw data managed by the system (read corresponding identifier belongs to *RId*) as required data. This makes the third requirement too severe, because some of the raw data managed by the system will never have to be retrieved, but it at least ensures that retrieval is possible. The second problem encountered is "How to check if data can be re-created from other raw data?". Raw data originally created from other raw data by a number of tool invocations (e.g. of synthesis tools or editors), can be re-created from these data, if all the information regarding how this was done is still available. This is typically information about which tools were invoked, the order in which these invocations occurred and which input data were supplied during these invocations. This type of information is managed by the design methodology management system. Therefore, the third requirement can only be checked by looking at both the design data store state and the design methodology management system.

4.1.6 Commands

Like a digital system data store, a design data store will feature commands to read and to change the contents of its state. For data retrieval we introduce the "Read" command, which results in the data selected by the corresponding "address" to be retrieved. Execution of this command will not change the DDS state and will present the results to the environment via the "design data output". For changing the contents of the DDS state we introduce the "Add", "Delete" and "Replace" commands. The "Add" and "Delete" commands result in data being added to and deleted from the DDS state, respectively. The "Replace" command will result in data stored by the DDS being replaced by new data. The location at which the design data store state is changed is determined by the corresponding address. The data to be added to the store by either the "Add' or "Replace" commands are received via the "design data input".

4.1.7 Addresses

Addresses are used to (al)locate the data manipulated by the DDS commands. Addresses do this by selecting an element of the DDS state. How are these addresses defined? Since the DDS state is a relation, this question can be rephrased to: How can we address (read "select") a relation element? As we have seen, relations are either represented by a set or a sequence, relating their elements in an unordered or ordered way, respectively. For the DDS state relation these elements are either raw data, raw data identifiers, attributes or other relations, which again are represented by a set or sequence. Relation elements can be addressed using a statement about the properties the selected elements are required to have. Sequence elements can additionally be addressed by making a statement about their location in this sequence. We formalise this as follows

Definition 4.1.3 *Set address*

For a set S a *set address* is a predicate A, which addresses those elements $el \in S$ for which $A(el)$ holds.

Definition 4.1.4 *Sequence address*

For a sequence T a *sequence address* is a predicate A, which addresses those elements $el = T_{pos}$, for which $A(el, pos)$ holds.

To illustrate this, consider the set representing the raw data identifier characterisation relation "AttrRIdMap" used in Equation 4.30, which is given by

$$\{<<\text{ALU, OmaDes}>, \text{spec}>, <<\text{ALU, OmaDes}>, \text{opt}>, <<\text{library, OmaLib}>, \text{lib}>, <<\text{ALU, OmaRep}>, \text{rep}>\}. \tag{4.31}$$

If we want to select the set element containing the key and raw data identifier of the OMA report data, then this can be done using the following set address

$$A_{\text{Rep}}(el) = (el_1 =< \text{ALU}, \text{OmaRep} >). \tag{4.32}$$

Using this address, the sequence <<ALU, OmaRep>, rep> can be selected, because it is the only relation element with key <ALU, OmaRep>. If we want to select the raw data identifier "rep" of this sequence, then this can be done by employing the following sequence address

$$A_{\text{RIdVal}}(el, pos) = (pos = 2). \tag{4.33}$$

which selects the raw data identifier of a raw data identifier characterisation relation element. Note that the selection of this identifier is based purely on its position. This is very often the case for sequence addresses.

Some addresses result in the selection of more than one element. To illustrate this consider the following address

$$A_{\mathrm{Des}}(el) = (el_1 =< \mathrm{ALU}, \mathrm{OmaDes} >). \tag{4.34}$$

When applied to the raw data identifier characterisation relation of Equation 4.31, this address will result in the selection in two elements, namely <<ALU, OmaDes>, spec> and <<ALU, OmaDes>, opt>.

We will group the relation elements selected by a set or sequence address using a set referred to as a *selection set*, which is defined as follows.

Definition 4.1.5 *Selection set*

For a relation R and an address A the *selection set* $R(A)$ contains those relation elements for which A holds. For a relation represented by a set and addressed by a set address this set is defined by

$$R(A) = \{el | el \in R \wedge A(el)\}.$$

Similarly, the selection set of a sequence addressed by a sequence address is given by

$$R(A) = \{el | \exists pos : el = R_{pos} \wedge A(el, pos)\}.$$

The selection set representing the results of using address A_{Des} of Equation 4.34 to select elements of relation 4.31 is given by

$$\{<<\mathrm{ALU}, \mathrm{OmaDes}>, \mathrm{spec}>, <<\mathrm{ALU}, \mathrm{OmaDes}>, \mathrm{opt}>\}. \tag{4.35}$$

Using set and sequence addresses, only the top level elements of a nested relation can be selected. If we want to address lower level elements of such a relation, the situation becomes a little bit more complicated. To illustrate this consider the following DDS state relation, containing the DDS relation of Equation 4.30, which is defined by

$$State = <Raw, RId, Attr, Rel>, \tag{4.36}$$

where *Rel* is given by

$$\begin{aligned}
&Rel = \\
&\quad \{<\mathrm{AttrRIdMap}, \mathcal{P}((Design \times Type) \times RId), \\
&\qquad \{<<\mathrm{ALU}, \mathrm{OmaDes}>, \mathrm{spec}>, <<\mathrm{ALU}, \mathrm{OmaDes}>, \mathrm{opt}>, \\
&\qquad\ <<\mathrm{library}, \mathrm{OmaLib}>, \mathrm{lib}>, <<\mathrm{ALU}, \mathrm{OmaRep}>, \mathrm{rep}>\}>, \\
&\quad\ <\mathrm{Version\text{-}of}, \mathcal{P}(OmaDes \times OmaDes), \{<\mathrm{opt}, \mathrm{spec}>\}>, \\
&\quad\ <\mathrm{RIdRawMap}, \mathcal{P}(RId \times Raw), \\
&\qquad \{<\mathrm{spec}, Spec>, <\mathrm{opt}, Opt>, <\mathrm{lib}, Lib>, <\mathrm{rep}, Rep>\}>\}.
\end{aligned}$$

If we want to address "rep" in *State*, than this can no longer be done using a single sequence address. The reason for this is that "rep" is not an element of the DDS state sequence. Relation elements like "rep" can be addressed using a sequence of set or sequence addresses. The first step involved in addressing "rep" in this DDS state, is the selection of the relation set *Rel* from the state sequence. This can be achieved using the following sequence address

$$A_{\mathrm{Rel}}(el, pos) = (pos = 4). \tag{4.37}$$

After *Rel* has been selected, the raw data identifier characterisation relation it contains can be addressed using the following predicate

$$A_{\mathrm{AttrRIdMap}}(el) = (el_1 = \mathrm{AttrRIdMap}). \tag{4.38}$$

The corresponding representation set *AttrRIdMap* can be selected from the resulting relation definition sequence <AttrRIdMap, (Design× Type)× RId, *AttrRIdMap*> using

$$A_{\mathrm{RelRepr}}(el, pos) = (pos = 3). \tag{4.39}$$

From this set "rep" can be selected using set address A_{Rep} of (see Equation 4.32) followed by sequence address A_{RIdVal} (see Equation 4.33).

So if we want to address "rep" in the state of Equation 4.36, than this can be done by a sequence of set and sequence addresses given by

$$< A_{\mathrm{Rel}}, A_{\mathrm{AttrRIdMap}}, A_{\mathrm{RelRepr}}, A_{\mathrm{Rep}}, A_{\mathrm{RIdVal}} >. \tag{4.40}$$

Relation addresses, so addresses which select an arbitrary relation element, can now be defined as follows

Definition 4.1.6 *Relation address*

A *relation address* is a sequence $< A_1, A_2, \ldots, A_n >$ of set or sequence addresses.

Which relation elements are selected by a relation address? A relation address represented by the sequence $< A_1, A_2, \ldots, A_n >$ will select all elements of the corresponding selection set $R(< A_1, A_2, \ldots, A_n >)$.

Definition 4.1.7 *Relation address selection set*

For a relation R and a relation address $< A_1, A_2, \ldots, A_n >$, the corresponding selection set $R(< A_1, A_2, \ldots, A_n >)$ is defined by

$$R(< A_1, A_2, \ldots, A_n >) = \bigcup_{R' \in R(A_1)} R'(< A_2, \ldots, A_n >), \text{ and}$$

$$R(\epsilon) = \{R\}.$$

This recursive definition states that the selection set of a relation address is obtained by the unification of a number of other selection sets, which are obtained by application of the reduced sequence $< A_2, \ldots, A_n >$ to the elements selected from R by A_1. The recursion stops when the address sequence is reduced to the empty sequence ϵ. When ϵ is used to address a relation (element) R, then this will result in the selection of R itself.
As an example consider the selection set

$$\mathit{State}(< A_{\text{Rel}}, A_{\text{AttrRIdMap}}, A_{\text{RelRepr}}, A_{\text{Des}}, A_{\text{RIdVal}} >), \tag{4.41}$$

which contains all "ALU" representations of type "OmaDes" stored in the state *State* of Equation 4.36. It is obtained as follows

$$\begin{aligned}
&\mathit{State}(< A_{\text{Rel}}, A_{\text{AttrRIdMap}}, A_{\text{RelRepr}}, A_{\text{Des}}, A_{\text{RIdVal}} >) \\
&\quad = \mathit{Rel}(< A_{\text{AttrRIdMap}}, A_{\text{RelRepr}}, A_{\text{Des}}, A_{\text{RIdVal}} >) \\
&\quad = < \text{AttrRIdMap}, \mathcal{P}((\mathit{Design} \times \mathit{Type}) \times \mathit{RId}), \mathit{AttrRIdMap} > \\
&\qquad\qquad\qquad\qquad (< A_{\text{RelRepr}}, A_{\text{Des}}, A_{\text{RIdVal}} >) \\
&\quad = \mathit{AttrRIdMap}(< A_{\text{Des}}, A_{\text{RIdVal}} >) \\
&\quad = << \text{ALU}, \text{OmaDes} >, \text{spec} > (< A_{\text{RIdVal}} >) \\
&\qquad \cup << \text{ALU}, \text{OmaDes} >, \text{opt} > (< A_{\text{RIdVal}} >) \\
&\quad = \text{spec}(\epsilon) \cup \text{opt}(\epsilon) \\
&\quad = \{\text{spec}\} \cup \{\text{opt}\} \\
&\quad = \{\text{spec}, \text{opt}\}.
\end{aligned} \tag{4.42}$$

Set and sequence addresses can only be used to select elements of grouping and sequencing relations, respectively. When addressing relation elements using a sequence of such addresses, an error will occur if one of these addresses is applied to a relation of the wrong type. This can be formalised as follows

Definition 4.1.8 *Valid relation address*

A relation address $< A_1, A_2, \ldots, A_n >$ is said to be *valid* with respect to a relation R if and only if for every address prefix $< A_1, A_2, \ldots, A_{i-1} >$ $(1 \leq i \leq n)$ the following holds

$\forall R' \in R(< A_1, A_2, \ldots, A_{i-1} >)$:
R' is a grouping relation if and only if A_i is a set address $\wedge$
R' is a sequencing relation if and only if A_i is a sequence address.

We conclude this section by introducing some (shortcuts for) frequently employed addresses. An example of a class of such addresses are the relation representation selection addresses. For a relation identified by *RelId*, the address $A_{\mathit{RelId}\text{Repr}}$ can be used to select the corresponding representation. This address is defined by

$$A_{RelId\text{Repr}} = < A_{\text{Rel}}, A_{RelId}, A_{\text{RelRepr}} >, \tag{4.43}$$

where

$$A_{RelId}(el) = (el_1 = RelId). \tag{4.44}$$

An example of such an address is $A_{\text{AttrRIdMapRepr}}$, which addresses the representation of the raw data identifier characterisation relation.

The type of a relation *RelId* can be selected using the address $A_{RelId\text{Type}}$. This address is defined by

$$A_{RelId\text{Type}} = < A_{\text{Rel}}, A_{RelId}, A_{\text{RelType}} >, \tag{4.45}$$

where

$$A_{\text{RelType}}(el, pos) = (pos = 2.) \tag{4.46}$$

Another frequently used class of addresses are the raw data identifier selection addresses. For an attribute key *key*, the address $A_{\text{Id}(key)}$ will select the raw data identifier characterised by this key and is defined by

$$A_{\text{Id}(key)} = < A_{\text{Rel}}, A_{\text{AttrRIdMap}}, A_{\text{RelRepr}}, A_{key}, A_{\text{RIdVal}} >, \tag{4.47}$$

where

$$A_{key}(el) = (el_1 = key). \tag{4.48}$$

As an example consider $A_{\text{Id}(<\text{ALU,OmaDes}>)}$, which when applied to the DDS state of Equation 4.36, results in the selection set {spec,opt}.

Sometimes addressing should result in all the relation elements being selected. For a relation represented by a set S this can be done using the address A_{True} which, when applied to S, results in the following selection set

$$S(A_{\text{True}}) = S. \tag{4.49}$$

The address sequences presented above often form the bases for the creation of new address sequences. New sequences can be formed by combination of a number of existing addresses and/or address sequences. For n address sequences $AS_1, AS_2, \ldots, AS_n$ the sequence

$$< AS_1, AS_2, \ldots, AS_n > \tag{4.50}$$

is used to represent the new address sequence

$$< AS_{1_1}, AS_{1_2}, \ldots, AS_{1_{|AS_1|}}, AS_{2_1}, AS_{2_2}, \ldots, AS_{2_{|AS_2|}}, \ldots, AS_{n_1}, AS_{n_2}, \ldots, AS_{n_{|AS_n|}} > \tag{4.51}$$

In this definition addresses are considered as address sequences with length 1.

As an example consider the following alternative definition of $A_{\text{Id}(key)}$ (see Equation 4.47), which defines this address reusing the address sequence $A_{\text{AttrRIdMapRepr}}$ defined by Equation 4.43 and which is given by

$$A_{\text{Id}(key)} = < A_{\text{AttrRIdMapRepr}}, A_{key}, A_{\text{RIdVal}} >. \tag{4.52}$$

4.1.8 Status information

During the operation of the design data store, the following things can go wrong:

- The command issued is neither "Read", "Add", "Delete" nor "Replace",
- the data to be stored is not of the correct type, or
- the address is not valid with respect to the design data store state.

This can be signalled to the environment of the design data store by producing status information like "Invalid command", "Data type mismatch" and "Invalid address", respectively. Correct operation can signalled by generating a message like "Ok".

4.1.9 The behaviour of the design data store

As shown in Figure 4.3, the behaviour of the design data store is determined by its *output function OF* and its *next state function NSF*, which are used to read and change the state of the DDS, respectively. Both functions take two arguments, namely the input and the current state of the design data store. The store input consists of three components, namely the command to be performed, the corresponding address and the data to be stored. It can be represented by a triple $< Command, Address, DataIn >$. The store output consists of two components, namely the data which was retrieved and the status information produced. It will be represented by a pair $< DataOut, Status >$. For a current state *State* the output function can now be defined as follows.

$$OF(<Command, Address, DataIn>, State) = \tag{4.53}$$

$$\begin{cases} < State(Address), \text{“Ok”} > & \text{if } Command = \text{“Read”} \\ & \text{and } ValidAddr(Address, State) \\ <\bot, \text{“Invalid command”}> & \text{if } \neg ValidCom(Command) \\ <\bot, \text{“Data type mismatch”}> & \text{if } \neg ValidData(Command, DataIn, Address, State) \\ <\bot, \text{“Invalid address”}> & \text{if } \neg ValidAddr(Address, State) \\ <\bot, \text{“Ok”}> & \text{otherwise} \end{cases}$$

So when issued the "Read" command and supplied with a valid address, the output function will return the data selected from the DDS state by *Address*. When an error occurs or when data are being stored rather than retrieved, the *DataOut* component of the output function result will be "undefined", which is indicated using the $\bot$ symbol. In these cases, the *Status* output component will indicate the reason for this absence of data.

In the definition of *OF* a number of predicates have been introduced, namely *ValidCom*, *ValidData* and *ValidAddr*, which detect if an error has occurred. These predicates are defined as follows. The predicate *ValidCom* determines if its argument *Command* is a valid command and is defined by

$$\mathit{ValidCom}(\mathit{Command}) = (\mathit{Command} \in \{\text{Read}, \text{Add}, \text{Delete}, \text{Replace}\}). \tag{4.54}$$

The predicate *ValidData* checks if the datum *DataIn* to be stored is of the correct type. If the address *Address* selects an element of the representation of a relation *RelId*, denoted by the fact that $A_{\mathit{RelId}\text{Repr}}$ is a prefix of *Address*, then after storage of *DataIn* *RelId* should still be of the correct type. Therefore *ValidData* is defined as follows

$$\begin{aligned}\mathit{ValidData}(\mathit{Command}, \mathit{DataIn}, \mathit{Address}, \mathit{State}) = \\ \mathit{State}'(A_{\mathit{RelId}\text{Repr}}) \in \mathit{State}(A_{\mathit{RelId}\text{Type}}),\end{aligned} \tag{4.55}$$

where *State'* the new state after storage of *Data*, which is given by

$$\mathit{State}' = \mathit{NSF}(< \mathit{Command}, \mathit{Address}, \mathit{DataIn} >, \mathit{State}). \tag{4.56}$$

The function *ValidAddr* determines if *Address* is a valid address in the context of a DDS state *State*.

The execution by a design data store of the commands "Add", "Delete" and "Replace" will change its state. Since this state is represented by a relation, changes to this state can be modelled by a relation transformation. We will introduce three relation operators for representing the state changes effectuated by these commands. For the addition to, deletion of and replacement of relation elements we define the *addition operator* $.[. + .]$, the *deletion operator* $.[/.]$ and the *replacement operator* $.[./.]$, respectively. When applied to a relation R, a relation element E and an address A, the addition operator will result in a new relation, denoted by $R[E + A]$, which is obtained from R by the addition of E at the location specified by A. The deletion operator will remove the relation elements selected by an address A from a relation R. The resulting relation is denoted by $R[/A]$. When applied to a relation R, a relation element E and a relation address A, the replacement operator will produce a new relation $R[E/A]$, which is obtained from R by replacing the data stored at the locations which A specifies by E. In Appendix B the addition, deletion and replacement operators are formally defined. In this chapter we will introduce these operators by giving some examples of how these are typically employed.

As an example of how the addition operator is used, consider how the decision to additionally monitor the "Version-of" relation affects a DDS state *State*, which features the DDS relation of Equation 4.29. Firstly, the "Version-of" definition sequence is added to the DDS relation *Rel* transforming *State* to the new state *State'* defined by

$$\mathit{State}' = \mathit{State}[< \text{Version-of}, \mathcal{P}(\mathit{OmaDes} \times \mathit{OmaDes}), \{\} > +A_{\text{Rel}}]. \tag{4.57}$$

After this, every derivation of a new design description from an existing one will result in the corresponding pair being added to the representation of the "Version-of" relation. For example, addition of the pair $<$ opt, spec $>$ will transform *State'* to the state presented in Equation 4.36 using the following transformation

$$\mathit{State}'[< \text{opt}, \text{spec} > +A_{\text{Version-ofRepr}}]. \tag{4.58}$$

As an example of how the deletion operator is used consider the transformation

$$State\,[/ < A_{\text{Rel}}, A_{\text{Version-of}} >], \tag{4.59}$$

which shows how the "Version-of" relation can be removed from the state *State* of Equation 4.36, if monitoring of this relation is no longer required.

As an example of how the replacement operator is typically employed, consider a situation where the raw data identifier "opt" is replaced by a new identifier "optn". Every occurrence of "opt" in the "Version-of" relation of a DDS state *State* can be changed to "optn" using the following state transformation

$$State\,[\text{optn}/A_{\text{Version-ofOpts}}], \tag{4.60}$$

where the address $A_{\text{Version-ofOpts}}$, which selects all the occurrences of "opt" as either the first or second element of the "Version-of" relation pairs, is given by

$$A_{\text{Version-ofOpts}} = < A_{\text{Version-ofRepr}}, A_{\text{True}}, A_{\text{SelectOpt}} >, \tag{4.61}$$

where

$$A_{\text{SelectOpt}}(el, pos) = (el = \text{opt}). \tag{4.62}$$

Using the addition, deletion and replacement operators, the next state function of the design data store can now be defined as follows.

$$NSF(< Command, Address, DataIn, >, State) =$$

$$\begin{cases} State & \text{if } Command = \text{Read} \\ State\,[DataIn + Address] & \text{if } Command = \text{Add} \\ State\,[/Address] & \text{if } Command = \text{Delete} \\ State\,[DataIn\,/\,Address] & \text{if } Command = \text{Replace} \end{cases} \tag{4.63}$$

Using the output and next state functions, we can now define the behaviour of the design data store itself. The operation of a design data store, whose current state is given by ss_1, can be modelled by a function *dds*, defined by

$$dds(< si_1, si_2, \ldots, si_n >) = < so_1, so_2, \ldots, so_n >, \tag{4.64}$$

which maps an arbitrary sequence of design data store inputs $< si_1, si_2, \ldots, si_n >$ to the resulting design data store output sequence $< so_1, so_2, \ldots, so_n >$. The relation between the elements of the input and the corresponding output sequence is given by

$$so_i = OF(si_i, ss_i) \text{ for } 1 \le i \le n\ , \tag{4.65}$$

where

$$ss_i = NSF(si_{i-1}, ss_{i-1}) \text{ for } 2 \le i \le n. \tag{4.66}$$

4.1.10 Concurrency control

Design data management systems are typically multi-user systems, i.e., they are able to service multiple users simultaneously. As a consequence, the corresponding design data store should also be a multi-user system. In this case, the DDS model presented above, in which a design data store is only able to process one command at a time, is no longer sufficient. In this thesis we are not going to present an explicit model of the much more complex multi-user design data store. The reason is that even for a multi-user store the behaviour with respect to each individual command can still be described in terms of the output function presented in Equation 4.53 and the next state function of Equation 4.63. What is different is the DDS state to which these functions are applied. A DDS processes its inputs by first using the address to select some of its state elements, followed by the application of the corresponding command to the results. Both steps take time, during which the state can be changed due to the concurrent processing of other DDS inputs. Therefore, the DDS state encountered during the execution of a DDS input will in general not be equal to the state at the moment at which its execution commenced. Moreover, this state is strongly affected by the concurrency control mechanism [BK91] [Bre95] [vHtBvLvW94] which the design data store employs.

A major advantage of our decision to split our DDMS model into a control interpreter which handles complex designer interactions and a design data store which features a much simpler interface is that it greatly facilitates discussions about concurrency control mechanisms. To be more specific, instead of having to consider complex control inputs, we only have to consider the effect of four much simpler commands, namely "Read", "Add", "Delete" and "Replace", which operate on a DDS state represented by a single relation.

When concurrent access to a design data store is allowed, this gives rise to a number of problems. For example, the following situations are considered to be problematic:

- A state element is in the process of being changed at the moment another command tries to access it, or
- a state element is still being accessed at the moment it is deleted or replaced.

In both situations the results produced by the commands accessing the state element will be undefined.

A possible solution for the problems described above is to construct a DDS which behaves according to the "first come, first served" principle. In the case of a DDS "first come" has to be interpreted as "first accessed". State elements are accessed during the evaluation of address sequences. Suppose we are evaluating an address sequence $< A_1, A_2, \ldots, A_n >$ for a DDS state *State*. In the first step the *State* itself will be accessed. In the second step all the state elements selected by A_1 are accessed, so all elements of $\mathit{State}(< A_1 >)$. In the next steps all the elements selected by $\mathit{State}(< A_1, A_2 >)$, $\mathit{State}(< A_1, A_2, A_3 >)$, ... and $\mathit{State}(< A_1, A_2, \ldots, A_n >)$ are accessed. So during the evaluation of an address sequence all the elements of the following set will be accessed

$$State(\epsilon) \cup State(< A_1 >) \cup State(< A_1, A_2 >) \cup \ldots \cup State(< A_1, A_2, \ldots, A_n >). \quad (4.67)$$

The "first come, first served" principle boils down to a concurrency control mechanism which blocks access to certain state elements until other evaluations which previously accessed these elements have been finished. In case a grouping or sequencing relation has been addressed with the purpose to add, delete or replace a relation element, all access to this relation will be blocked. This prevents the first access problem describe above from occurring. Addition of relation elements can always be performed, even if the other relation elements are operated upon concurrently. In case of deletion or replacement of a relation element, the second access problem can occur. The "first come, first served" principle prevents this by stalling the deletion or replacement of relation elements until all the DDS input evaluations which previously accessed these elements have been completed.

Another problem which occurs when concurrent access to the DDS is allowed is that the relation between DDS inputs and the resulting DDS outputs is lost. This problem can be solved by associating a unique identifier with each DDS input. This identifier can then be used to identify the resulting DDS output.

The DDS state is not only changed by the design data store itself. If a state element, e.g. a piece of raw data, is read from the store it can be changed outside the store, thereby indirectly changing the DDS state. This is not a desirable situation, because in this case the DDS has no means to control concurrent access to these state elements. A simple solution for this problem is to return a copy of the selected relation element rather than the element itself. Because this copy is not a part of the DDS state, changes to it will not affect the DDS state, while on the other hand changes to the original state element will not affect this copy. The drawback of this approach is that it results in a lot of data copying. If this is a problem, other, more complicated, solutions can be employed. For example, if only read access is required, copying can be avoided by making the data read-only as long as it is used outside the store. Another possibility is to block access to the data from inside the store as long as it is manipulated externally.

4.2 The control interpreter

The design data store discussed in the previous section provides a very simple interface to its environment. In contrast to this, design data management systems will in general feature a more complex and higher level interface. We model this by the introduction of a control interpreter. The *control interpreter* (see figures 4.3 and 4.4) interprets the DDMS input and translates it into a sequence of simpler design data store inputs. The value of a store input occurring in this sequence is determined by both the DDMS input and the results produced by the design data store in response to the store inputs preceding it. The control interpreter will use the resulting design data store output sequence to form the DDMS output.

To illustrate how the control interpreter operates consider a situation in which we want to add a new workspace to a workspace hierarchy. A DDMS can be instructed to do this

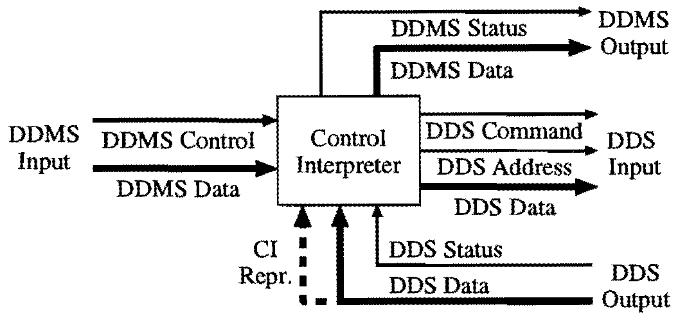


Figure 4.4: The DDMS control interpreter.

by giving it a command like "addWorkspace(*name, parent, owner, type*)", which instructs it to create a new workspace *name* of type *type* which is owned by *owner* and which is a child of the workspace *parent*. In response to this the DDMS will change its organisation to reflect the resulting new workspace hierarchy. We model this by a control interpreter receiving a control input representing the "addWorkspace" command. In response to this the control interpreter will generate a sequence of DDS inputs, which change the DDS state in such a way that the "Workspace" relation of Equation 4.21, the "WorkspaceHier" relation of Equation 4.22, and the "Workspaces" relation of Equation 4.26 represent the new workspace hierarchy.

4.2.1 Control inputs

The operation of the control interpreter is guided by its control input. We introduce control inputs using a subclass of control inputs referred as *data queries*. Data queries specify a scheme to retrieve information from the design data store. The bases of a data query is formed by a number of addresses. Additionally, a data query features a *retrieval function* which, when applied to the selection sets specified by these addresses, will return the data to be retrieved.

The simplest form of data query is given by a single address combined with a retrieval function equal to the identity function. The result of this data query is the selection set of this address. Such a data query can be represented by a pair $< I, A >$, where I and A are the identity function and the address, respectively. For a design data store state *State* this query will return

$$I(State(A)) = State(A). \tag{4.68}$$

As an example consider the query $< I, A_{\mathrm{Id}(<\mathrm{ALU},\mathrm{OmaDes}>)} >$, which when applied to the DDS state of Equation 4.36, returns the identifiers of all ALU representations of type

"OmaDes" from the raw data identifier characterisation relation. The result of this query is given by the selection set

$$\mathit{State}\,(A_{\mathrm{Id}(<\mathrm{ALU,Des}>)}) = \{\mathrm{opt}, \mathrm{spec}\}. \tag{4.69}$$

If we want to know how many "ALU" representations of type "OmaDes" are being managed by the DDMS, rather than what the actual identifiers are, then this number can be acquired using the data query $< \mathit{Size}\,, A_{\mathrm{Id}(<\mathrm{ALU,Des}>)} >$. The unary function *Size* used in this query returns the number of elements in its argument set. The result is obtained as follows

$$\mathit{Size}\,(\mathit{State}\,(A_{\mathrm{Id}(<\mathrm{ALU,Des}>)})) = \mathit{Size}\,(\{\mathrm{opt}, \mathrm{spec}\}) = 2. \tag{4.70}$$

Queries can be based on more than one address. Suppose we want to retrieve the identifiers characterised by attribute key <ALU, OmaDes>, of which the corresponding raw data were used as a specification to generate an optimised version of this representation, but which were themselves not created by optimisation of another design (i.e. were synthesised from a higher level design or directly created using an editor). The identifiers of the "ALU" representations of type "OmaDes" can be selected using address $A_{\mathrm{Id}(<\mathrm{ALU,Des}>)}$. The fact that a piece of raw data identified by "opt" is an optimisation of another piece of raw data identified by "spec", is represented in the DDS state by the occurrence of the pair $< \mathrm{opt}, \mathrm{spec} >$ as one of the elements of the "Version-of" relation. Therefore, the set of all specification and optimisation identifiers can be selected by the A_{Spec} and the A_{Opt} address, respectively, which are defined by

$$A_{\mathrm{Spec}} = < A_{\mathrm{Version\text{-}ofRepr}}, A_{\mathrm{True}}, A_{\mathrm{isSpec}} >, \text{ and} \tag{4.71}$$

$$A_{\mathrm{Opt}} = < A_{\mathrm{Version\text{-}ofRepr}}, A_{\mathrm{True}}, A_{\mathrm{isOpt}} >, \tag{4.72}$$

where

$$A_{\mathrm{isSpec}}(el, pos) = (pos\ = 2), \text{ and} \tag{4.73}$$

$$A_{\mathrm{isOpt}}(el, pos) = (pos\ = 1). \tag{4.74}$$

When applied to the DDS state of Equation 4.36, these addresses will result in the selection sets {spec} and {opt}, respectively.

The desired identifiers can now be retrieved using the following query

$$< \mathit{IntSub}\,, A_{\mathrm{Id}(<\mathrm{ALU,Des}>)}, A_{\mathrm{Spec}}, A_{\mathrm{Opt}} >. \tag{4.75}$$

The function *IntSub* used in this query, calculates the intersection of the selection sets corresponding to the first two addresses, and subsequently subtracts the elements of the third address' selection set from the resulting set. It is defined by

$$\mathit{IntSub}\,(S_1, S_2, S_3) = (S_1 \cap S_2) \setminus S_3. \tag{4.76}$$

The result of the query is obtained as follows

$$\begin{aligned} &(\mathit{State}\,(A_{\mathrm{Id}(<\mathrm{ALU,Des}>)}) \cap \mathit{State}\,(A_{\mathrm{Spec}})) \setminus \mathit{State}\,(A_{\mathrm{Opt}}) = \\ &(\{\mathrm{opt}, \mathrm{spec}\} \cap \{\mathrm{spec}\}) \setminus \{\mathrm{opt}\} = \\ &\{\mathrm{spec}\}. \end{aligned} \tag{4.77}$$

In general, a data query is defined as follows.

Definition 4.2.1 *Data query*

A *data query* is a n-tuple $< RetrFunc, A_1, A_2, \ldots, A_{n-1} >$, where *RetrFunc* the *retrieval function* and A_1, A_2, ..., A_{n-1} the *query addresses. RetrFunc* is a function with $n-1$ arguments. For a DDS state *State*, the query returns the result of the application of *RetrFunc* to the selection sets $State(A_1)$, $State(A_2)$, ..., $State(A_{n-1})$ corresponding to the query addresses.

Suppose we want to retrieve all raw data of type "OmaDes" representing an "ALU". The corresponding identifiers can be obtained using the query $< I, A_{\text{Id}(<\text{ALU},\text{Des}>)} >$. The desired raw data can now be retrieved by selection of the raw data associated with these identifiers by the raw data identification relation "RIdRawMap". A natural way to specify this is to use a *nested query.* A nested query is a query which is defined in terms of the results of other queries. For our example, the address which selects the raw data from "RIdRawMap" is defined in terms of the identifier query. The nested query which retrieves all raw data of type "OmaDes" representing an "ALU" is given by

$$< I, A_{\text{RawALUDes}} >, \tag{4.78}$$

where

$$A_{\text{RawALUDes}} = < A_{\text{RIdRawMapRepr}}, A_{\text{ALUDes}}, A_{\text{RawVal}} >, \tag{4.79}$$

$$A_{\text{ALUDes}}(el) = (el_1 \in < I, A_{\text{Id}(<\text{ALU},\text{Des}>)} >) \text{ , and} \tag{4.80}$$

$$A_{\text{RawVal}}(el, pos) = (pos = 2). \tag{4.81}$$

The control interpreter executes data queries in a number of steps. First it produces a sequence of design data store inputs, one for each query address, which results in the corresponding selection sets being returned by the design data store. Secondly, the control interpreter will produce the query result by combining these selection sets using the retrieval function. Finally, it will presents this result to the DDMS environment via the DDMS output.

As stated before, the *control interpreter* translates the DDMS input into a sequence of simpler design data store inputs and subsequently combines the resulting design data store outputs to form the DDMS output. This process is controlled by the DDMS control input. By programming the control interpreter the DMS developers determine which control inputs it will accept and how these are to be translated into a sequence of design data store inputs. No matter which language is used to described control inputs, their semantics is given by

Definition 4.2.2 *Control input*

A *control input* is a pair $< IGF, OGF >$, where *IGF* the *DDS Input Generation Function* and *OGF* the *DDMS Output Generation Function.*

Figure 4.5 shows the data produced and/or used by the control interpreter and the design data store during the control input interpretation process. The DDS input generation function *IGF* of a control input defines which DDS input sequence *DDSIS* the control interpreter will generate in response to this input. The input sequence produced depends on the value of the DDMS data input *DataIn* and the DDS output sequence *DDSOS* produced in response to *DDSIS*, so

$$IGF(DataIn, DDSOS) = DDSIS. \tag{4.82}$$

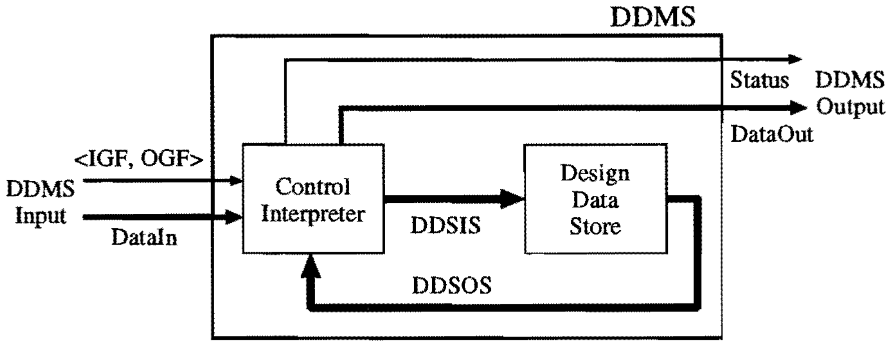


Figure 4.5: The control input interpretation process.

The value of a store input occurring in *DDSIS* will not depend on the complete DDS output sequence, but rather on that part of this sequence produced by the design data store in response to the store inputs preceding it. This is reflected by the following statement

$$IGF(DataIn, DDSOS)_i = IGFC(DataIn, DDSOS[1, i-1]), \tag{4.83}$$

indicating that *IGF* can be defined in terms of a function *IGFC* (C stands for causality), which produces the $i - th$ DDS input based on the value of *DataIn* and the first $i - 1$ elements of the DDS output sequence.

The DDMS output generation function *OGF* of a control input defines the DDMS output generated by the control interpreter as a result of its interpretation. The DDMS output produced depends on the value of the DDMS data input *DataIn* and the DDS output sequence *DDSOS*. The DDMS output can be represented by a pair $< DataOut, Status >$, defined by

$$OGF(DataIn, DDSOS) = < DataOut, Status >. \tag{4.84}$$

Since data queries are control inputs, it is possible to represent a query Q, given by $< RetrFunc, A_1, A_2, \ldots, A_{n-1} >$, by a control input pair $< IGF_Q, OGF_Q >$. The DDS input generation function IGF_Q is defined by

$$IGF_Q(DataIn, DDSOS) = \\ << \text{Read}, A_1, \perp>, < \text{Read}, A_2, \perp>, \ldots, < \text{Read}, A_{n-1}, \perp>>. \quad (4.85)$$

Notice that the DDS input sequence generated by this function is independent from the DDS output sequence *DDSOS*. For a design data store, which is in state ss_1, the DDS output sequence produced in response to this input sequence is given by

$$\begin{aligned} &DDSOS \\ &= dds(< \text{Read}, A_1, \perp>, < \text{Read}, A_2, \perp>, \ldots, < \text{Read}, A_{n-1}, \perp>) \quad (4.86) \\ &= << ss_1(A_1), Status_1 >, < ss_1(A_2), Status_2 >, \ldots, < ss_1(A_{n-1}), Status_{n-1} >>. \end{aligned}$$

The output generation function OGF_Q describes how the DDS output is used to produce the query result. It is given by

$$\begin{aligned} &OGF_Q(DataIn, \quad (4.87) \\ &\quad <<ss_1(A_1), Status_1 >, < ss_1(A_2), Status_2 >, \ldots, < ss_1(A_{n-1}), Status_{n-1} >>) = \\ &\begin{cases} < \text{RetrFunc}(ss_1(A_1), ss_1(A_2), \ldots, ss_1(A_{n-1})), \text{“Ok”} > \\ \qquad\qquad \text{if } \forall i : 1 \le i < n : Status_i = \text{“Ok”} \\ <\perp, < Status_1, Status_2, \ldots, Status_{n-1} >> \quad \text{otherwise} \end{cases} \end{aligned}$$

This function represents that if the DDS data selected by the query addresses are successfully read, indicated by a corresponding status equal to “Ok”, the query result is obtained by applying RetrFunc to these data. If on the other hand one or more errors occur during this DDS data retrieval, then the query result will be undefined and the reason for this failure will be indicated using the DDS status sequence.

Nested queries can not be represented using the control input pair $< IGF_Q, OGF_Q >$. Some of the addresses used in such a query depend on the results of other queries and therefore on the contents of the design data store. Since the DDS input sequence produced by IGF_Q does not depend on the corresponding DDS output sequence, IGF_Q is not able to produce the input sequence required for nested queries. For representing nested queries an input generation function is required which depends on the value of *DDSOS*. As an example consider the nested query $< I, A_{\text{RawALUDes}} >$ presented by Equation 4.78. The control input pair equivalent $< IGF_{RAD}, OGF_{RAD} >$ is defined by

$$IGF_{RAD}(DataIn, DDSOS) = \\ << \text{Read}, A_{\text{Id}(<\text{ALU},\text{Des}>)}, \perp>, < \text{Read}, RawAddrGen(DDSOS_1), \perp>>, \quad (4.88)$$

where the raw address generation function *RawAddrGen* is defined by

$$RawAddrGen(< DataOut, Status >) = \\ < A_{\text{RIdRawMapRepr}}, A_{\text{ALUDes}}, A_{\text{RawVal}} > \quad (4.89)$$

and the address A_{ALUDes} is given by

$$A_{\text{ALUDes}}(el) = el_1 \in DataOut. \quad (4.90)$$

4.2.2 The behaviour of the control interpreter

The behaviour of the control interpreter can be modelled by a function ci, which when applied to its three arguments, consisting of the control input pair, the DDMS data input and the DDS output sequence, produces the DDS input sequence and the value associated with the DDMS output. This function is defined by

$$ci(< IGF, OGF >, DataIn, DDSOS) = < DDSIS, < DataOut, Status >>, \tag{4.91}$$

where the output pair is given by

$$< DDSIS, < DataOut, Status >> = \\ < IGF(DataIn, DDSOS), OGF(DataIn, DDSOS) >. \tag{4.92}$$

4.2.3 Concurrency control

Like the design data store, the control interpreter will in general allow concurrent access. This means that different control inputs are processed simultaneously. The resulting design data store input sequence will be a mixture of the DDS inputs generated by the control inputs currently being processed. Since the DDS concurrently evaluates the available inputs, the resulting DDS outputs will additionally be mixed.

The first problem encountered is that the relation between the control inputs and the corresponding design data store inputs and outputs is lost. A possible solution is to have the control interpreter associate each of the DDS inputs with an unique identifier consisting of the control input producing it and its position in the corresponding DDS input sequence. Since the design data store will associate this identifier with the resulting output, the DDS output sequence of a control input can now be reconstructed from the total DDS output sequence.

Another problem is that the concurrent evaluation of the different DDS inputs can lead to incorrect results. This even occurs when only one control input is handled at a time. Since the corresponding DDS inputs are processed concurrently, there is no guarantee that, at the moment the DDS store starts the evaluation of an input, the processing of the preceding inputs has finished. If the proper processing of a DDS input depends on the completion of some of the inputs preceding it, then this can lead to incorrect results. This can however easily be avoided. A possible way to this is to use a DDS input generation function, which makes the generation of such an input dependent on the presence of the DDS outputs resulting from the evaluation of these preceding inputs. This can even be used if these results do not actually determine the input's value.

The situation becomes a lot more complicated when the control interpreter is processing different control inputs simultaneously. In this case there is no way to predict beforehand which DDS inputs will be processed concurrently. This leads to problems if the proper processing of a control input requires the part of the DDS state it operates on to be unchanged by other control inputs during the time of its evaluation. A possible remedy is to read the complete DDS state part involved and perform the operations on the resulting private copy rather than on the actual DDS state. If required, the DDS state can be updated later on by replacing the original design part with its modified copy.

4.3 The behaviour of the DDMS

The behaviour of a design data management system is a result of the interaction of its two components: The design data store and the control interpreter. Therefore, if we want to describe its behaviour in terms of a function *ddms*, then it should be possible to define this function in terms of the functions *dds* of Equation 4.64 and *ci* of Equation 4.91, which represent the behaviour of the corresponding design data store and control interpreter, respectively. The design data management system function *ddms* determines what, for a given control input pair and DDMS input value, the corresponding DDMS output value will be. It is defined by

$$ddms(< IGF, OGF >, DataIn) = < DataOut, Status >. \tag{4.93}$$

The DDMS output pair $< DataOut, Status >$ is given by the second component of the pair generated by the control interpreter function *ci*, so

$$< DataOut, Status > = ci(< IGF, OGF >, DataIn, DDSOS)_2. \tag{4.94}$$

The DDS output sequence *DDSOS* used by the control interpreter for the generation of the DDMS output is produced by the design data store, which is represented by

$$DDSOS = dds(DDSIS), \tag{4.95}$$

where the DDS input sequence *DDSIS* is given by the first component of the pair produced by *ci*, so

$$DDSIS = ci(< IGF, OGF >, DataIn, DDSOS)_1. \tag{4.96}$$

4.4 Summary

In this chapter a detailed design data management system model was presented, which describes the system in terms of two interacting components: A design data store and a control interpreter. The design data store contains the raw data produced during the design process, the associated meta data and will be used to keep track of the relations existing between these data. In this chapter a detailed description was given of how the design data store organises its data, how these data can be stored and how these data can retrieved afterwards. In addition, some restrictions were presented, which a design data store should satisfy in order to guaranty that it will operate properly. The control interpreter interprets the abstract control signals it receives from the DDMS's environment by translating these into a sequence of simpler design data store operations. In this chapter it was demonstrated how these control signals can be represented and how the control interpreter performs the translation of these signals into a sequence of design data store inputs. Finally, a description was given of how the interaction of the design data store and the control interpreter results in the behaviour of the total design data management system.

Chapter 5

The design tool management system

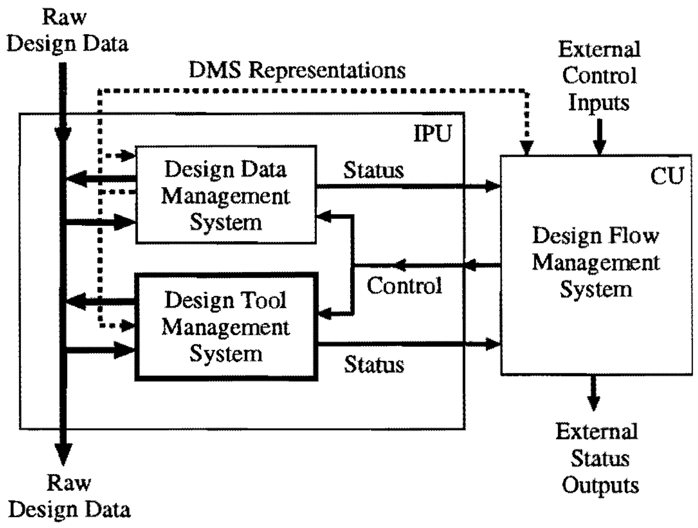


Figure 5.1: Environment of the design tool management system.

Tools are software programs assisting designers in performing their design task. A Design Tool Management System (DTMS) helps the designers making proper use of these tools. It should enable its users to refer to these tools logically rather than physically. This means that the system should abstract from tool invocation related information. On the other hand the system should supply the users with information about what the abilities of a certain tool are, about the types of its in- and outputs and how to fine-tune a tool for a specific application. In this chapter we will present a model of a design tool management system satisfying these requirements. Figure 5.1 shows how the DTMS is positioned within the design management system.

The way the design tool management system handles its tools strongly resembles the way raw data are treated by the design data management system. Both raw data and tools are large pieces of data of which the actual contents are of no interest for design management purposes. Both raw data and tools are stored by the corresponding management system and can be retrieved by the users of these systems. Both the design data and design tool management systems contain additional information which is used to characterise the raw data and tools they manage. The basic difference between the DDMS and the DTMS is that the design tool management system has to execute the tools it stores. So in addition to the information used for tool selection, it should also contain information about tool invocation and feature a component enabling it to execute tools. Based on the considerations presented above we arrive at the DTMS structure depicted in Figure 5.2.

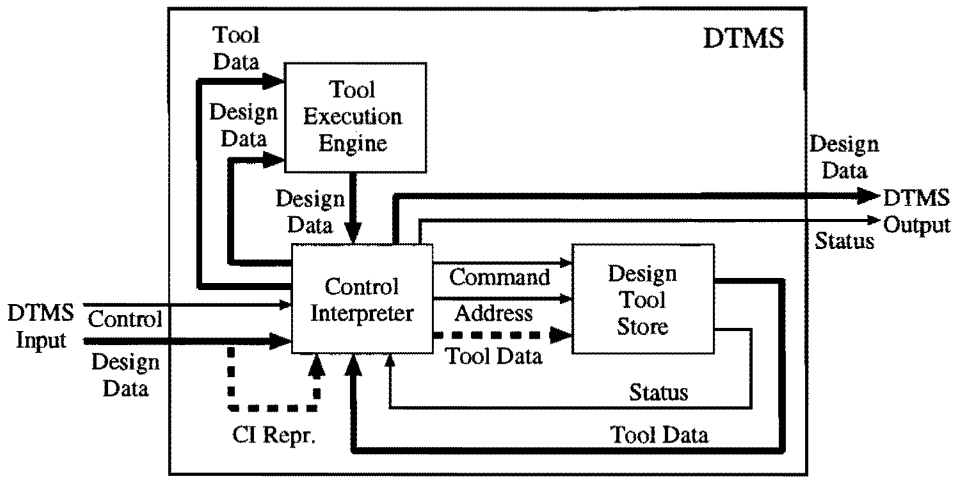


Figure 5.2: The design tool management system model.

In this figure the DTMS is modelled as consisting of three interacting components: a design tool store (DTS), a control interpreter (CI) and a tool execution engine (TEE). The *design tool store* only differs from the design data store by what is being stored rather than how this is done. The *tool execution engine* produces new design data by invocation of the tools retrieved from the design tool store on design data obtained from the design management store. Like the DDMS, the design tool management system features a *control interpreter*. This control interpreter not only supervises the interactions with the design tool store and the generation of the DTMS output, but also provides the tool execution engine with the tool and design data it operates upon and processes the resulting tool outputs.

Tools are created by tool developers. From the viewpoint of tool developers the tool descriptions they create are raw data to be managed by the design data management system. It is the task of the tool integrators to retrieve these tool descriptions from the DDMS and to store these at the appropriate address in the design tool store. Tools descriptions arrive at the DTMS via the design data input and are subsequently presented to the design tool store via the "Tool Data" bus. In Figure 5.2 this bus is represented by a dashed arrow, which indicates that it is only to be used by the design management system developers and not by the designers.

In Section 5.1 the design tool store is presented. Since it only differs from the design data store with respect to what is being stored, this treatment can be limited to a description of what the design tool store typically contains. The tool execution engine is discussed in Section 5.2. Finally, in Section 5.3 the control interpreter is introduced.

5.1 The design tool store

In analogy to the design data store, the following architecture for the design tool store is proposed (see Figure 5.3). Except for the state, the DTS is identical to the DDS. The *design tool store state* is determined by the tools and meta data currently stored by the DTS. In addition to the tool descriptions created by the tool developers, the store will also contain tool abstractions. Tool abstractions are created by tool integrators and can be used by designers to execute a tool without having to know the tool invocation details. In addition, tool developers will add meta data to the store characterising the stored tools and tool abstractions. These characterisation can be used by designers to select the appropriate tool. In the rest of this section we will discuss tool abstractions, tool characterisation and tool selection, respectively.

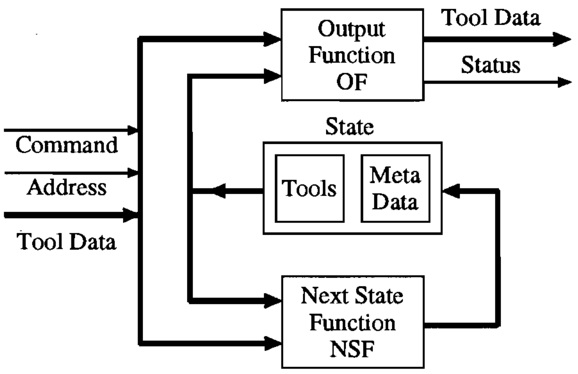


Figure 5.3: General structure of the design tool store.

5.1.1 Tool abstractions

As stated before, a design tool management system should enable its users to refer to its tools logically rather than physically. Therefore, from the user viewpoint, the tools managed by such a system should be *logical tools* rather than *physical tools*. A physical tool can be turned into a logical tool by creating a *tool abstraction*. Tool abstractions *abstract* from tool invocation information, enabling designers to execute a physical tool without the need to know the details of how this is done.

In our discussion about tool abstractions, we use the tools employed during the OMA timing driven matching process as an example. In fact, only one physical tool is used, namely OMA, but it is used to perform two completely different tasks, i.e., optimisation and matching. In analogy to the design flow management system of Nelsis [BtBvW92], which splits a tool into a set of activities representing the different design functions of this tool, we introduce the optimisation tool *Optimise* and the matching tool *Match* as the logical tools managed by the design tool management system rather than OMA itself.

The behaviour of the logical tool *Optimise* can be represented by a function, which when applied to a specification design *spec*, returns an optimised version *opt* of this design. This function is implemented by creating a tool abstraction which defines *Optimise* in terms of the corresponding physical tool OMA [Rov94]. Tool abstractions are often defined using a shell script. As an example consider the following UNIX C-shell script, which implements the behaviour of *Optimise* by running OMA in optimisation mode.

```
set type = `OmaInType spec`
cp spec spec.$type
ControlMod -mode opt -type $type omaCtrTmpl > spec.oma_ctr          (5.1)
oma spec.oma_ctr
mv spec.oma_out opt
```

This shell script does not define the *Optimise* function using mathematics, but in terms of a sequence of UNIX shell commands. The script assumes that there exists a file named "spec" containing the specification to be optimised. As a result of its execution, it will produce a file named "opt" containing the optimised design. The shell commands produce this optimised design in the following way. When OMA is to be applied to a design named *Design* expressed in the input format *Format*, it assumes its design and control inputs to be stored in files named *Design.Format* and *Design*.oma_ctr, respectively. The *Optimise* tool on the other hand only features one input design expressed in one of the input formats accepted by OMA. So before OMA can be invoked, we first have to determine what the type of the input file is, copy the input file to a file with the proper type extension and generate the required control file. The input file type is determined by the program "OmaInType" used in the first UNIX command. The result returned by this program is made available to the other commands by assigning it to the shell variable "type". The second command uses this variable to create the design input file required by OMA. The control file is generated from the template control file "omaCtrTmpl" using the control file modifier program "ControlMod". This program is invoked by the third UNIX command, which modifies "omaCtrTmpl" by setting the mode to "opt" and the input file type to "type". Because of the name conventions for its input files, OMA requires only one argument, i.e., its control file. OMA is executed by the fourth UNIX command. When invoked on a design named *Design*, OMA will store the optimised design in a file named *Design*.oma_out. Since we want this design to be stored in the file *opt*, the last command will move the optimised design to this file.

Tool inputs and outputs which are pieces of raw data, like *spec* and *opt* from the *Optimise* tool, can be accessed by the UNIX commands using a file with the same name. But what about other types of in- or outputs? As an example consider the *Match* tool. Its behaviour can be represented by a function, which when applied to a triple <*spec, lib, delaySpec*>, produces the pair <*net, delayRes*>. It maps the logic expressions of a logic level design *spec* onto a gate level netlist *net* using the gates specified in the library *lib*. This process can be fine-tuned by specifying the maximum delay *delaySpec* the resulting gate level netlist is required to have. Besides the netlist, it will also return the resulting maximum delay *delayRes*. We enable the UNIX commands used in the corresponding tool abstraction to access tool in and outputs like *delaySpec* and *delayRes*, which are natural numbers rather than raw data, by introducing a shell variable for each of these. As an example consider the following C-shell script which can be used to implement the *Match* tool.

```
set type = ‘OmaInType spec‘
cp spec spec.$type
ControlMod -mode match -constr $delaySpec -type $type omaCtrTmpl > spec.oma_ctr
oma spec.oma_ctr                                                          (5.2)
mv spec.oma_net net
set delayRes = ‘MaxDelay spec.oma_lis‘
```

This script only differs from the one used for *Optimise* by the control file which is employed and in terms of the results produced. The control modifier "ControlMod" creates a control file which puts OMA in "match" mode with a timing constraint equal to the value of the "delaySpec" variable. When used to match a design named *Design*, OMA will store the resulting netlist in a file named *Design*.oma_net and information about the matching process, e.g., the obtained maximum delay, in the corresponding report file *Design*.oma_lis. To obtain the outputs of *Match* the file "spec.oma_net" is moved to the file "net" and the maximum delay is extracted from the report file using the "MaxDelay" program, after which it is bound to the shell variable "delayRes".

5.1.2 Tool characterisation

A tool is characterised by information about what its abilities are, the types of its in- and outputs and how to fine-tune it for a specific application. This information is added to the design tool store by the tool integrators in the form of meta data. In this section we show examples of how tools can be characterised. To keep our discussion independent from the way characterisation information is represented in the design tool store, we will use mathematics rather than meta data to represent this information.

One way to characterise a tool is to represent it in terms of a relation, which describes how the tool inputs are mapped onto the corresponding outputs. As an example consider the relation representing the operation of the *Optimise* tool, which when invoked on a design produces an optimised version of this design. This relation is declared by

$$Optimise \subset \{< spec, opt > | spec \in OmaDes \wedge opt \in OmaDes\} \ . \tag{5.3}$$

It is defined by a set of pairs $< spec, opt >$, representing all possible input/output combinations for the optimisation tool.

Unlike *Optimise*, many tools will feature more than one input or output. These tools are modelled by a relation which maps the elements of the set of all possible input combinations to the corresponding elements of the set of all output combinations. As an example consider the *Match* tool. The relation representing this tool is declared by

$$\begin{aligned} Match \subset \{<< spec, lib, delaySpec >, < net, delayRes >> | \\ < spec, lib, delaySpec > \in OmaDes \times OmaLib \times \mathbb{N} \wedge \\ < net, delayRes > \in OmaNet \times \mathbb{N}\} \ . \end{aligned} \tag{5.4}$$

The optimised design *opt* produced by the optimisation tool *Optimise* is uniquely determined by the specification *spec*. Therefore, the corresponding relation will contain only one pair in which *spec* occurs as the first element and the corresponding optimised design *opt* can be obtained by taking the second element of this pair. This is not true for all tools. To illustrate this consider an editor. Given the original design and the user input, the resulting edited design is completely determined. However, modelling the complex user interactions of such an editor using a single relation is a very difficult task. Fortunately, for most design management purposes it is not necessary to explicitly model these interactions. It often suffices to represent the editor's operation in term of a relation which maps the original design onto all the designs which can be derived from it using the editor. As a result the original design will occur as the first element in a number of relation pairs, namely one for each possible edited design. Given such a relation and the original design, it is not possible to determine what the result of the editing operation will be, because this is determined by external influences which are not modelled by the relation.

For most design management purposes, it is not necessary to create a detailed model of a tool's operation; it often suffices to define the relation type rather than the relation itself. For tool selection however, some information is required about how a tool relates its inputs to its outputs. As an example consider the following characterisations of the *Optimise* and *Match* tools. Suppose there exist two functions *Func* and *ExprSize*. When applied to a design the function *Func* will return a canonical representation of its functionality, i.e., two designs des_1 and des_2 will be functionally equivalent if and only if

Func(*des*$_1$) = *Func*(*des*$_2$) holds. For logic level designs the function *ExprSize* will return a number giving an indication about the number and complexity of the logic expressions used in such a design. Using these functions the optimisation tool *Optimise* can be characterised by stating that

$$\begin{aligned} Optimise \subset \{<&spec, opt> \mid \\ & ExprSize(opt) < 0.6 * ExprSize(spec) \wedge \\ & Func(opt) = Func(spec)\} \ , \end{aligned} \tag{5.5}$$

representing that the optimisations performed by *Optimise* will typically result in a 40% reduction of the logic expression size and that it does this without changing the corresponding functionality. The *Match* tool will also not change the functionality of a design and will try to obtain a maximum delay smaller than the specified delay. This is represented by stating that

$$\begin{aligned} Match \subset \{<&< spec, lib, delaySpec >, < net, delayRes >> \mid \\ & Func(net) = Func(spec) \wedge \\ & delayRes \leq delaySpec\} \ . \end{aligned} \tag{5.6}$$

The results of a tool invocation will be stored by the design data management system. However, data storage is only useful if these data can be retrieved afterwards. To enable data retrieval, data have to characterised. A tool invocation will relate the tool's outputs to the corresponding inputs. For example, the application of *Optimise* to its *spec* input will result in an optimised *version* of this design, which is given by the output *opt*. This should be reflected in the design data management system by the addition of these designs to the "Version-of" relation. We represent this fact by stating that $< opt, spec > \in$ *Version-of*. Likewise, the fact that the netlist *net* produced by the *Match* tool *implements* the logical level design specification *spec* can be represented by stating that $< net, spec >$ $\in$ *Implements*.

5.1.3 Tool selection

Tool selection is largely determined by the tool abilities. The abilities of a tool can be specified by making a statement about the properties of the outputs produced by the tool. These properties are often expressed in terms of the properties of the corresponding inputs. As an example consider the ability specification of the *Optimise* tool, which is obtained by combination of the statements of Equation 5.3 and Equation 5.5, and is given by

$$\begin{aligned} Optimise \subset \{<&spec, opt> \mid \\ & spec \in OmaDes \wedge opt \in OmaDes \wedge \\ & ExprSize(opt) < 0.6 * ExprSize(spec) \wedge \\ & Func(opt) = Func(spec)\} \ . \end{aligned} \tag{5.7}$$

Tool selection based on tool abilities is guided by a statement about the properties the tool is required to have. Like tools, tool requirements can be represented by a relation. For example, the requirement *Req* can be represented by the relation

$$\begin{aligned} Req = \{ &< spec, opt > \mid \\ & spec \in OmaDes \wedge opt \in OmaDes \wedge \\ & ExprSize(opt) < 0.7 * ExprSize(spec)\} \ , \end{aligned} \tag{5.8}$$

which specifies that a tool has to be selected able to optimise an input design of type "OmaDes" such that the resulting output, which has to be of the same type, exhibits a logic expression size which is at least 30% smaller than that of the original design.

A tool will satisfy a requirement if and only if its relation is a subset of the requirement relation. For instance, the *Optimise* tool satisfies requirement *Req*, because the following statement holds

$$Optimise \subset Req \ , \tag{5.9}$$

which can be proven using the tool characterisation of *Optimise* presented in Equation 5.7.

Tool selection based on abilities will often produce more than one tool satisfying the requirement. A possible way to select between these alternatives is by considering the costs of the tool invocation. Tool costs can be expressed in a number of ways. Possible indicators are the computing time required for running the program, the amount of memory needed and the amount of money which has to be paid for each tool invocation. It is impossible to give an exact specification of the run time and memory requirements of a tool. The values of these properties will depend on some measure of the size of the tool inputs. For most tools this relation is not known. However, sometimes, the order of this dependency is known. For instance, by specifying that the memory requirements are of the order

$$O(ExprSize(spec)) \ ,$$

we represent that there is a linear dependency between the expression size of the design input *spec* and memory needed for running *Optimise*.

5.2 The tool execution engine

Although design tool management systems have a number of provisions to assist designers during tool selection and to hide tool invocation related information, all this will be useless if it will not result in the execution of tools. The tools managed by a DTMS are invoked by the *tool execution engine*. As an example consider the invocation of the *Match* tool used during the overspecified timing constraints step of the OMA timing driven matching process. When supplied via the design data input with a triple $< Opt, Lib, 0 >$ (where *Opt* the result of the optimisation step, *Lib* a gate library, and "0" a timing constraint equal to zero nanoseconds), and the *Match* tool abstraction via the tool data input, the tool execution engine will produce the corresponding output, e.g., a pair $< Net, 199 >$, where *Net* a gate netlist and "199" the corresponding maximum delay.

5.3 The control interpreter

Like the control interpreter of the design data management system, the *DTMS control interpreter* (CI) translates the DTMS input into a sequence of simpler design tool store inputs and will generate the DTMS output. However, unlike the DDMS, the generated DTMS output is not obtained by combining the resulting store outputs. Instead the construction of this output is based on the design data produced by the tool execution engine. The tool execution engine generates these data by invoking the tool descriptions retrieved from the design tool store. This process is controlled by the DTMS control input. A design tool management system will feature a language to describe these control inputs. Control inputs describe one of the possible interaction patterns of the three DTMS components by specifying the values of the CI outputs in terms of the data received via the CI inputs. Although the control interpreter outputs in principle depend on all the CI inputs, in practice this will not be the case. This is reflected by the following definition.

Definition 5.3.1 *DTMS control input*

A *DTMS control input* is a triple $< IGF, TIGF, OGF >$, where *IGF* the *DTS Input Generation Function*, *TIGF* the *TEE Input Generation Function*, and *OGF* the *DTMS Output Generation Function.*

Figure 5.4 shows the data produced and/or used by the control interpreter, the design tool store, and the tool execution engine during the DTMS control input interpretation process. The DTS input generation function will produce a DTS input sequence *DTSIS*. The values of the elements of this input sequence, not only depend on the DTS output sequence *DTSOS* produced in response to the previous elements, but also on the tool execution engine output *TEEO*. This is represented by stating that

$$IGF(DTSOS, TEEO) = DTSIS. \tag{5.10}$$

The reason why we make the design tool store input sequence dependent on the results of the tool execution engine, is that sometimes invocation of a tool will not produce the desired results. For instance, if the matcher *Match* has been selected to match a design with a maximum delay equal to 100ns and the resulting netlist has a delay of 119ns, then it should be possible to select another tool to do the job. By making *IGF* dependent of *TEEO*, it becomes possible to query the DTS for another tool if the previously selected tool fails.

The TEE input generation function will provide the tool execution engine with its input *TEEI*. The tool execution engine input is represented by a pair <Tool, ToolInput>, e.g., the pair $< Match, < Opt, Lib, 0 >>$ used during the OTC matching step of the OMA timing driven matching process. The TEE input generation function takes the tool description

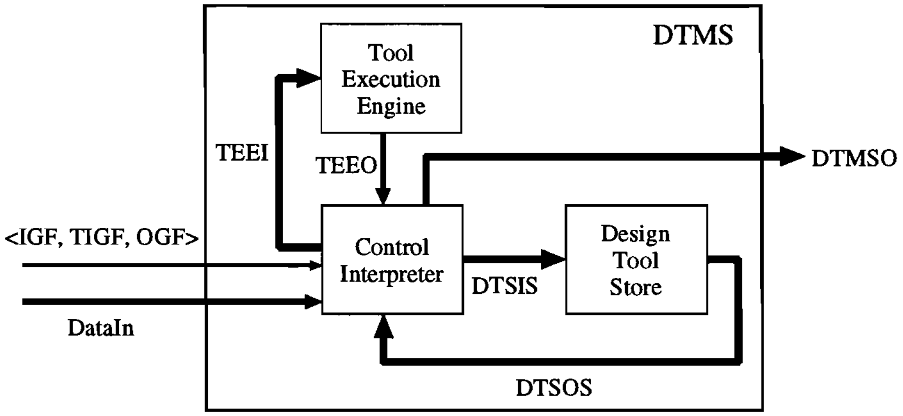


Figure 5.4: The DTMS control input interpretation process.

from the DTS output *DTSO* and generates the corresponding tool input using the data received via the DTMS design data input *DataIn* or the data *TEEO* produced by the TEE during the previous tool invocation. This is represented by

$$TIGF(DataIn, TEEO, DTSO) = TEEI \; . \tag{5.11}$$

By making *TIGF* dependent on the design data produced during the previous tool invocation, it becomes possible to generate the required design data using a sequence of tool invocations rather than a single one, each using the results of the previous run as their input.

The DTMS output generation function *OGF* uses the design data produced by the tool execution engine to generate the DTMS output *DTMSO*, so

$$OGF(TEEO) = DTMSO \; . \tag{5.12}$$

5.4 Summary

In this chapter a detailed design tool management system model was presented. The DTMS was modelled as consisting of three interacting components: a design tool store, a tool execution engine and a control interpreter. The design tool store of the DTMS only differs from the design data store of the design data management system with respect to what is being stored, i.e., tools rather than raw data. Therefore our treatment about the design tool store was limited to a description of what the design tool store typically contains. In this context we have discussed tool abstractions, tool characterisation and tool selection. Tool abstractions turn physical tools into logical tools, enabling designers to execute a tools without having to know about tool invocation details. The behaviour of a tool can be characterised by making statements about what the abilities of the tool are, about the types of its in- and outputs and how to fine-tune the tool for a specific application. Tools are selected from the design tool store based on their characterisation. The selected tools will be invoked on the corresponding design data by the tool execution engine. The operation of the design tool store and the tool execution engine is controlled by the control interpreter by interpretation of the DTMS's control input. We have demonstrated how these control inputs can be modelled.

Chapter 6

The design flow management system

When the designer's data and tools are managed by the design data and design tool management system, then the only task remaining for the designer is to decide how the required design can be obtained using these tools and data. This involves selection of the tools to invoke, determination of the order of these invocations and selection of the data to which these tools are to be applied. The *Design Flow Management System* (DFMS) assists the designers in making these decisions. It does this by executing design flow representations. These representations describe when which commands to issue to the design data and the design tool management system to obtain the required design. As shown in Figure 6.1 the design flow management system controls the DDMS and DFMS by issuing commands via the "Control" bus. The DDMS and DTMS report their progress to the DFMS by sending information via the "Status" bus.

In this chapter a detailed design flow management system model will be presented. The general structure of a DFMS can be represented as depicted in Figure 6.2. Like the design tool management system, the design flow management system is modelled as consisting of three interacting components: a *design flow store* (DFS), a *control interpreter* (CI), and a *flow execution engine* (FEE). The design flow store contains representations of the flows managed by the DFMS and flow related information. The flow representations are expressed in the flow description language supported by the flow execution engine. Flow descriptions can be retrieved from the DFS by sending flow queries to the control interpreter. The selected flows are send to the flow execution engine, where they are executed concurrently. We will refer to the flows currently executed by the FEE as *active flows*. The active flows control the operation of the rest of the design management system. For example, active flows enforce decisions regarding which tools are to be invoked on what data by sending the appropriate control inputs to the design data and design tool management systems via the control bus. They monitor the operation of these systems using the resulting status information they receive via the status bus. Furthermore, the active flows also take care of the interaction with the design management system's environment using the external control and status busses. They even control the activation of other flows, by issuing flow queries to the design flow store.

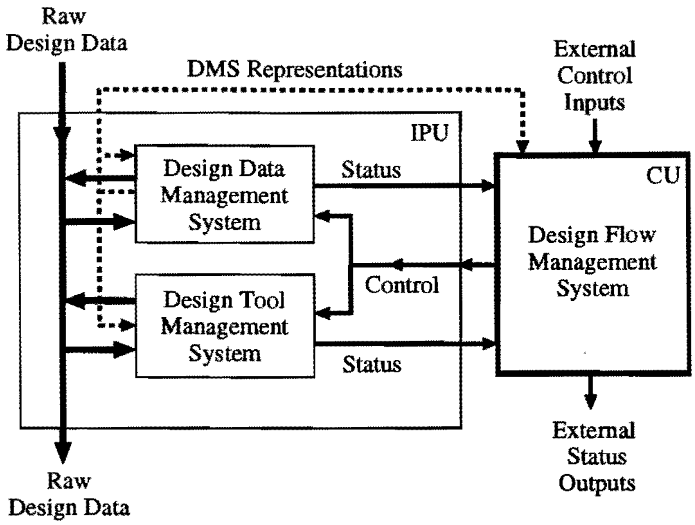


Figure 6.1: Environment of the design flow management system.

In Section 6.1 of this chapter design flows are introduced. The flow execution engine is discussed in Section 6.2. The control interpreter and design flow store of the DFMS are very similar to the interpreters and stores encountered in the previous two chapters, and will therefore not be treated here.

6.1 Design flows

Like the control unit of a digital system, the operation of the design flow management system can be modelled by a state machine. The input of this machine not only consists of the status signals produced by the design data and design tool management systems, but also of the external control signals its receives. The output it produces are either control inputs for the design data and the design tool management systems or external status signals. The behaviour of the design flow management system is determined by the design flow descriptions it executes concurrently, i.e., the active design flows. The operation of a flow execution engine executing a flow description can again be represented by a state machine. Unlike the total DFMS state machine, these state machines will not only interact with the DFMS environment, but also with the design flow store and with each other. Based on this state machine model, we now define an active design flow as follows.

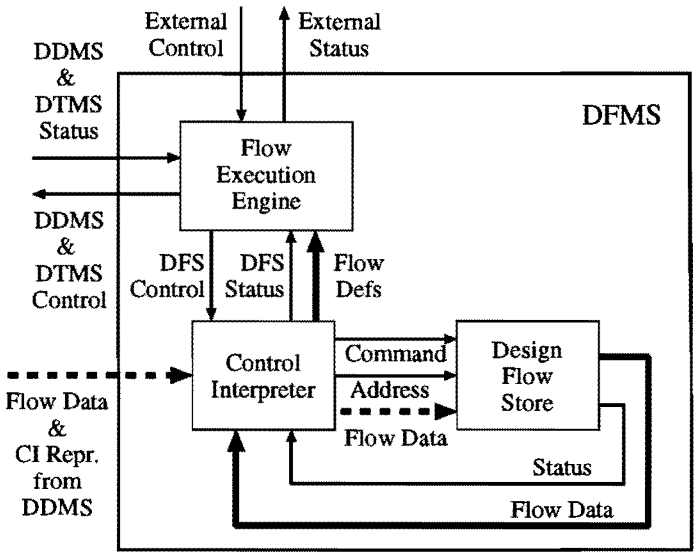


Figure 6.2: The design flow management system model.

Definition 6.1.1 *Active design flow*

An active design flow can be represented by a triple <*FId*, *CS*, *FlowDef*>, where

- *FId* - the flow identifier,
- *CS* - the current state of the flow, and
- *FlowDef* - the design flow definition.

An active flow is uniquely identified by its flow identifier *FId*. The current state *CS* is the dynamic part of an active flow. It changes in response to the flow input and therefore enables flows to make their actions dependent on events which occurred in the past. The operation of an active flow is defined by the corresponding design flow definition. A design flow definition *FlowDef* is represented by a 7-tuple $< S, S_0, F, E, BD, \lambda, \delta >$ where

- S - the state set,
- $S_0 \in S$ - the start state,
- $F \subset S$ - the final state set,
- $E \subset S$ - the error state set,

- BD - the bus definition set,
- λ - the output function, and
- δ - the next state function.

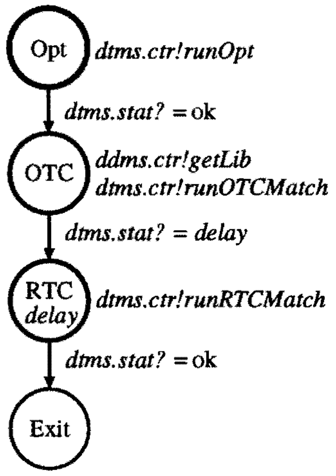


Figure 6.3: The OMA timing driven matching flow.

We will illustrate the different components of a design flow definition using the OMA Timing Driven Matching (OTDM) flow presented in Figure 6.3. The state set S defines the state space of a flow, i.e., all the potential current states. At first glance the OTDM flow depicted in Figure 6.3 appears to feature four states: The optimisation state "Opt", the overspecified timing constraints state "OTC", the realistic timing constraints state "RTC", and the exit state "Exit". In contrast to the other nodes, however, the node labelled "RTC" represents a set of states rather than a single state. The reason is that it features a parameter *delay*, which will contain the value of the maximum delay obtained during the overspecified constraints matching step. So during the realistic timing constraints matching step, the flow will be in a state represented by a pair $<\mathrm{RTC}, \mathit{delay}>$ ($\mathit{delay} \in \mathbb{N}$). As a result, the state set S of the OTDM flow is given by

$$\{\mathrm{Opt}, \mathrm{OTC}, \mathrm{Exit}\} \cup \{<\mathrm{RTC}, \mathit{delay}> | \mathit{delay} \in \mathbb{N}\} . \tag{6.1}$$

The moment a flow is activated, its current state will be equal to the start state S_0. The OMA timing driven matching flow starts in the optimisation state "Opt".

Flow execution ends when it reaches one of the final or one of the error states. When a final state is entered, e.g., "Exit" in case of the OTDM flow, this indicates that the flow execution has been successful. However, when an error state is reached, then this represents that a failure occurred. The information about whether a flow has been executed successfully or unsuccessfully is passed back to the flow who originally activated it, e.g., by sending it status information like "Ok" and "Error", respectively.

Unlike a digital system, for which the connections between its components are fixed, the busses used in our design management system model are created dynamically whenever they are required. Activation of a flow will result in the creation of the busses described in the bus definition set BD. For the OTDM flow this set contains the following elements: two busses which are used by the OTDM flow to control and monitor the design tool management system, which are given by

$$< \text{dtms.ctr}, \{\text{dfms.control}\}, \{\text{dtms.control}\}, OPB > , \tag{6.2}$$

$$< \text{dtms.stat}, \{\text{dtms.stat}\}, \{\text{dfms.stat}\}, OPB > , \tag{6.3}$$

a bus via which the flow controls the design data management system, defined by

$$< \text{ddms.ctr}, \{\text{dfms.control}\}, \{\text{ddms.control}\}, OPB > , \tag{6.4}$$

and the following three busses

$$< \text{des}, \{\text{ext.data.in}, \text{dtms.data.out}\}, \{\text{dtms.data.in}\}, OPB > , \tag{6.5}$$

$$< \text{lib}, \{\text{ddms.data.out}\}, \{\text{dtms.data.in}\}, OPB > \text{ , and} \tag{6.6}$$

$$< \text{net}, \{\text{dtms.data.out}\}, \{\text{ext.data.out}\}, OPB >, \tag{6.7}$$

of which bus "des" is used to get the original input design from the DMS's environment and to temporarily store the corresponding optimised design. Bus "lib" transports the cell library from the DDMS to the DTMS, and the bus "net" is used to present the resulting gate network to the environment. Note that the type of these busses is given by *OPB*, which denotes that these are all One Place Buffers (see Section 3.3.3).

After the busses of a flow have been created, execution of the flow will change their contents. The current state of the busses can be represented by a set

$$\{< cont, bd > | bd \in BD \wedge cont \in bd_{\text{ContType}}\} , \tag{6.8}$$

which contains a bus for each bus definition in BD and where the bus contents *cont* are of the type bd_{ContType} specified by bd. In the rest of this chapter we will use *dtms.ctr*, *dtms.stat*, *ddms.ctr*, *des*, *lib*, and *net* to refer to the elements of this set identified by "dtms.ctr", "dtms.stat", "ddms.ctr", "des", "lib", and "net", respectively. In addition, we will use B to refer to the set of all possible current bus state sets.

Given the flow's current state and the currently available status and external control information, the output function λ determines which commands the flow will issue. The current contents of the flow's busses determine which status and control information is available. Moreover, a flow issues its commands by sending messages to its busses, thereby changing the state of these busses. So the output function can be defined in terms of how it, given the current flow state and the current state of the busses, generates a new state for the busses. This is represented by stating that

$$\lambda \in S \times B \rightarrow B \ . \tag{6.9}$$

As an example, consider the output function of the OTDM flow, which is given by

$$\lambda(state, \emptyset) = \begin{cases} \{dtms.ctr!runOpt\} & \text{if } state = \text{Opt} \\ \{ddms.ctr!getLib, dtms.ctr!runOTCMatch\} & \text{if } state = \text{OTC} \\ \{dtms.ctr!runRTCMatch\} & \text{if } state =< \text{RTC}, delay > \end{cases} \ . \tag{6.10}$$

Note that we have only listed those elements of the current bus state set and the new bus state set, which are either read and/or changed by the output function. The fact that the OTDM output function does not depend on the current state of the busses, is represented by a current bus state set argument which equals the empty set. The fact that the new bus state sets returned by the OTDM output function only contain modified versions of the *dtms.ctr* and *ddms.ctr* busses, indicates that all other busses are left unchanged.

The output function of Equation 6.10 represents that the OTDM flow controls the DDMS and DTMS using the *getLib*, *runOpt*, *runOTCMatch*, and *runRTCMatch* control inputs. The control inputs described in Chapter 4 and Chapter 5 instruct the DDMS and DTMS how to generate the value for the design data and status outputs given the design data input value. However, since each of these in- and outputs may be connected to a number of busses, the control input model presented in these chapters no longer suffices. The control input concept should be extended with a mechanism for specifying the busses to be used. We solve this problem by considering control inputs as instructions of how to generate a new bus state set given the current bus state set. In practice this means that all references to the design data input are replaced by references to one or more of the busses connected to this input. In addition the result produced by the output generation function will no longer be a $<$ *design data, status* $>$ pair, but the new bus state set.

As an example consider the *runOpt* control input used in the output function of Equation 6.10. When the OTDM flow reaches the optimisation state "Opt", it will send this control input to the design tool management system. It will instruct the DTMS to invoke the logical tool *Optimise* on the original input design, which is received via the "ext.data.in" input and transported to the DTMS using the "des" bus. The resulting optimised design will be again be written to the "des" bus. The *runOpt* control input can be looked upon as a bus state set transformer, operating on the bus state subset $\{des, dtms.stat\}$, and is defined by

$$runOpt_{IGF}(DTSOS, TEEO) =<< \text{Read}, A_{\text{Tool}(optimise)} >> \ , \tag{6.11}$$

$$runOpt_{TIGF}(\{des\}, TEEO, DTSO) =< DTSO_1, des? > \text{ , and} \tag{6.12}$$

$$runOpt_{OGF}(TEEO) = \{des!TEEO, dtms.stat!\text{"ok"}\} . \tag{6.13}$$

The design tool store input generated by the function $runOpt_{IGF}$ selects the *Optimise* tool from the design tool store using the address $A_{\text{Tool}(optimise)}$. In general, $A_{\text{Tool}(tId)}$ can be used to select the tool description identified by tool identifier *tId*. The tool execution engine input generation function $runOpt_{TIGF}$ produces the TEE input by combining the description of the *Optimise* tool with the design representation read from the *des* bus. The output generation function $runOpt_{OGF}$ generates the new bus state set, which is obtained from the original set by writing the optimised design to the *des* bus and the status "ok" to the status bus *dtms.stat*.

When the OTDM flow reaches the overspecified timing constraints state "OTC", it will send the control input *getLib* to the design data management system. This control input instructs the DDMS to write the cell library addressed by A_{Lib} to the *lib* bus and is given by

$$getLib_{IGF}(\emptyset, DDSOS) = << \text{Read}, A_{\text{Lib}}, \perp >> \text{ , and} \tag{6.14}$$

$$getLib_{OGF}(\emptyset, DDSOS) = \{lib!(DDSOS_1)_1\} . \tag{6.15}$$

In addition the OTDM flow will send the *runOTCMatch* control input to the design tool management system, which will instruct the DTMS to invoke the logical tool *Match* on the design read from *des* using the library read from *lib* and a timing constraint equal to zero. The resulting maximum delay is send back to the flow using the *dtms.stat* bus. The *runOTCMatch* control input is defined by

$$\begin{aligned}
&runOTCMatch_{IGF}(DTSOS, TEEO) = << \text{Read}, A_{\text{Tool}(match)} >> , \\
&runOTCMatch_{OGF}(TEEO) = \{dtms.stat!TEEO_2\}, \text{ and} \\
&runOTCMatch_{TIGF}(\{des, lib\}, TEEO, DTSO) = < DTSO_1, < des?, lib?, 0 >> ,
\end{aligned} \tag{6.16}$$

where $TEEO_2$ the maximum delay achieved during the OTC step.

When the OTDM flow arrives at the realistic timing constraint state "RTC", it will issue the control input *runRTCMatch* to the DTMS, which instructs it to run the *Match* tool using the value of the *delay* parameter as the timing constraint. The resulting gate network is presented to the DMS's' environment using the *net* bus. The *runRTCMatch* control input is given by

$$\begin{aligned}
&runRTCMatch_{IGF}(DTSOS, TEEO) = << \text{Read}, A_{\text{Tool}(match)} >> , \\
&runRTCMatch_{OGF}(TEEO) = \{net!TEEO_1\}, \text{ and} \\
&runRTCMatch_{TIGF}(\{des, lib\}, TEEO, DTSO) = < DTSO_1, < des?, lib?, delay >> .
\end{aligned} \tag{6.17}$$

Proper execution of the *runOTCMatch* control input requires the cell library to be present at the *lib* bus. This library is retrieved from the design data store using the control input *getLib*. How do we guarantee that the DDMS has finished processing *getLib* before the DTMS starts execution of *runOTCMatch?*. A possible way to achieve this is to introduce an extra state "LIB". If we move the generation of the *getLib* control input to this new state and if we make the transition to the "OTC" state dependent on the completion of *getLib*, then the library will always be available the moment the "OTC" state is reached. A disadvantage of this approach is that we have to introduce extra states, whose only purpose it is to synchronise the operation of the DDMS and the DTMS.

An alternative way to achieve our goal is to make the execution of the control inputs data driven, i.e., the DDMS and DTMS only start the evaluation of a control input at the moment the busses contain the data it requires. For instance, *runOTCMatch* will be activated the moment both the optimised design and the cell library become available at the *des* and *lib* bus, respectively. When the OTDM flow reaches the "OTC" state the optimised design is already present at *des*. The cell library becomes available at the *lib* bus the moment *getLib* is completed. This event will trigger the execution of *runOTCMatch*.

The next state function δ determines what, given the current flow state and the state of the busses, the next state of the flow will be. Although execution of the next state function will not result in data being written to the busses, it can result in data being consumed, i.e., read and subsequently deleted from a bus. So besides the next state, the next state function will also produce a new bus state. This is represented by stating that

$\delta \in S \times B \rightarrow S \times B$.

As an example consider the next state function of the OTDM flow, which is given by

$$\delta(state, \{dtms.stat\}) = \tag{6.18}$$

$$\begin{cases} <\text{OTC}, \{dtms.stat-\}> & \text{if } state = \text{Opt and } dtms.stat? = \text{“ok”} \\ << \text{RTC}, delay >, \{dtms.stat-\}> & \text{if } state = \text{OTC and } dtms.stat? = delay \\ <\text{Exit}, \{dtms.stat-\}> & \text{if } state =< \text{RTC}, delay > \\ & \text{and } dtms.stat? = \text{“ok”} \end{cases} .$$

This function represents that, after the OTDM flow has send a control input to the design tool management system, it will stay in the same state until the DTMS has finished processing it. The OTDM flow monitors the operation of the DTMS using the *dtms.stat* bus, via which it either receives the message "ok" or the value for the *delay* parameter, each of which indicates successful termination of the DTMS. After such a DTMS status signal has been read, it will be removed from the bus. We do this to avoid problems, like mistaking the value "ok" produced by the previous tool run as the value to be assigned to the *delay* parameter. Deletion of the status values after these are read, is indicated by the second result produced by the next state function, which is the new bus state subset $\{dtms.stat-\}$.

In contrast to the control inputs, whose execution is data driven, we will assume the invocation of the output and next state functions of the flow to be event driven, i.e., they are executed if one of their arguments changes value. Since the OTDM output function only depends on the state of the flow, it will be executed once after each state transition. The OTDM next state function is evaluated after each change of either the state *state* or the bus *dtms.stat*.

6.2 The design flow execution engine

The active flows executed concurrently by the flow execution engine not only control the operation of the DDMS and DTMS; they also supervise the activation of other flows. In the first step of flow activation, one of the active flows will send a flow query to the control interpreter of the design flow store. If the query is successful, the control interpreter will send the resulting design flow description to the design flow input of the FEE. In response to this, the flow execution engine will determine a suitable flow identifier, and will use it in combination with the design flow description to activate the flow.

If active flows can only be created by other flows, then how was the first flow activated? The answer is that we introduce a special flow, referred to as the *start-up flow*, which is automatically activated every time the design management system is started up. An example of a useful start-up flow is a *login flow*. This flow will query the DMS users for their login name and their password. It will manage information about the rights of the DMS users and will use this to restrict their access to data, tools and flows.

Flows are not only useful for automating parts of the design process, they can also be used to enforce design policies. For example, by allowing a designer to select the OTDM flow but not the individual tools *optimise* and *match*, we enforce the OMA timing driven matching policy.

When a flow is activated, the flow execution engine will create the busses described by the corresponding design flow definition. However, if a design flow definition is executed for the second time and the first one is still active, then all these busses will already exist. A possible way to handle such a situation is to have flows share these multiple defined busses. Sometimes bus sharing is useful, because it provides an easy means for inter flow communication. However, if bus sharing is not required, it will only cause problems, because it mixes up data belonging to different flows. An alternative solution is to associate a new property with each of the busses of a design flow definition, of which the value is either "private" or "public". A public bus will not be created if it already exists at the moment of flow activation. For a private bus with identifier *busId*, flow activation will always result in the creation of a new bus named *flowId.busId*, where *flowId* the identifier of the resulting active flow.

6.3 Summary

In this chapter a design flow management system model was presented. The DFMS was modelled as consisting of three interacting components: a *design flow store*, a *control interpreter* and a *flow execution engine*. Since the design flow store and the corresponding control interpreter only differ from those presented in the two previous chapters by what they manage, i.e., flows instead of data or tools, these components were not discussed in this chapter. The flow execution engine executes flow representations retrieved from the design flow store. These flow representations are expressed in the flow description language supported by the flow execution engine. We have introduced the term *active flows* to refer to the flow descriptions currently executed by the FEE. These active flows control the operation of both the design data and the design tool management systems. Furthermore, the active flows also take care of the interaction with the design management system's environment. They even control the activation of other flows, by issuing flow queries to the design flow store. The operation of a FEE running a flow description was modelled by a state machine. This state machine was defined in terms of how it controls its environment by sending control messages via its busses. Which control messages are issued by the state machine is determined by both its state and the status messages available at its busses. We have illustrated the FEE model by describing a state machine modelling a FEE running a flow automating the OMA timing driven matching process.

Chapter 7

A design management system description language

The development time of a CAD framework is mainly used up by the creation of its design management system component. The design management systems of most of the existing CAD frameworks were programmed using a general purpose language. The development time of such a system can be greatly reduced if it is implemented using a dedicated design management system description language. When such a language is employed, a design management system can be described at a higher level of abstraction. As a result, it not only becomes easier to create design management system descriptions but also to maintain and extend these.

As far as we know, there are currently no suitable design management system description languages available. Therefore, in this chapter we show how such a DMS description language can be constructed. Using the design management system model presented in the previous four chapters as a guideline, we formulate the requirements this language has to satisfy in Section 7.1. The definition of a DMS description language is not a trivial task. Luckily there exists a class of languages based on hierarchical coloured Petri nets [Jen92] [vH94], which already have most of the required characteristics. Design management systems feature extensive interaction with their users and with the operating system. Hierarchical coloured Petri nets lack support for describing these interactions. Therefore, we have extended the Petri net concept with special interaction constructs, resulting in interactive hierarchical coloured Petri nets. These interactive Petri nets satisfy all our formulated requirements and are introduced in Section 7.2. In Section 7.3 the feasibility of our approach is demonstrated by showing how interactive Petri nets can be used to create executable design management system representations.

7.1 Language requirements

Since it is our intention to use the DMS description language to implement design management systems, the first requirement we formulate is: *The DMS description language should enable us to create executable representations.* In this case, the DMS representations can not only be used to document but also to automate the described design management activities.

Design management systems are typically multi-user systems and will often have to serve a number of users simultaneously. Therefore, the second requirement is: *The language should feature a form of concurrency suitable for the representation of multi-user systems.*

A design management system features extensive interaction with its users and with the operating system. Therefore, the third requirement is: *The language should have constructs for representing user and operating system interactions.*

Our fourth requirement is: *The DMS description language should provide support for decomposition.* Why do we require support for decomposition? Although design management systems can on an abstract level be described in terms of three distinct interacting components, i.e., the DDMS, the DTMS and the DFMS, in practice they are never implemented this way. Instead, just like digital system designers, DMS developers will implement the DMS by decomposing it into a hierarchy of smaller systems, each of which are described in terms of a simpler DDMS, DTMS and DFMS. The reason for this is twofold. Firstly, for real life design management systems the three DMS components will become too complex to be handled as a single unit by the DMS developers. Secondly, most DMS developers will have a distributed view of a DMS, i.e., they usually consider it as collection of much smaller systems, which group related DMS elements. As an example, consider the implementation of the workspace concept. Although it is possible to do this using a single design data management system, this will in general be a complicated task. It is easier to use an alternative approach, in which the workspaces are viewed as communicating DMS subsystems. As an example consider the DMS of Figure 7.1, which is decomposed in such a way that it reflects the workspace hierarchy of Figure 2.4. For each of the workspaces it features a DMS subsystem, which groups all the data managed by the corresponding workspace. Its busses allow these systems to communicate in the way dictated by the workspace hierarchy.

There exists a class of languages, referred to as hierarchical languages, which provide extensive support for decomposition by featuring special constructs which allow designers to describe hierarchical designs. Hierarchical languages usually feature a module instantiation mechanism to introduce hierarchy. *Modules* are design parts with a well-defined interface. Modules are either primitive or composite. Primitive modules, e.g., the "ProjectWS", "GroupWS" and "PrivateWS" modules of Figure 7.1, form the leaves of the design hierarchy and are complete descriptions of a certain part of the design. Composite modules, on the other hand, are defined using instantiations of other modules as building blocks. As an example consider the "DMS" module of Figure 7.1. It consists

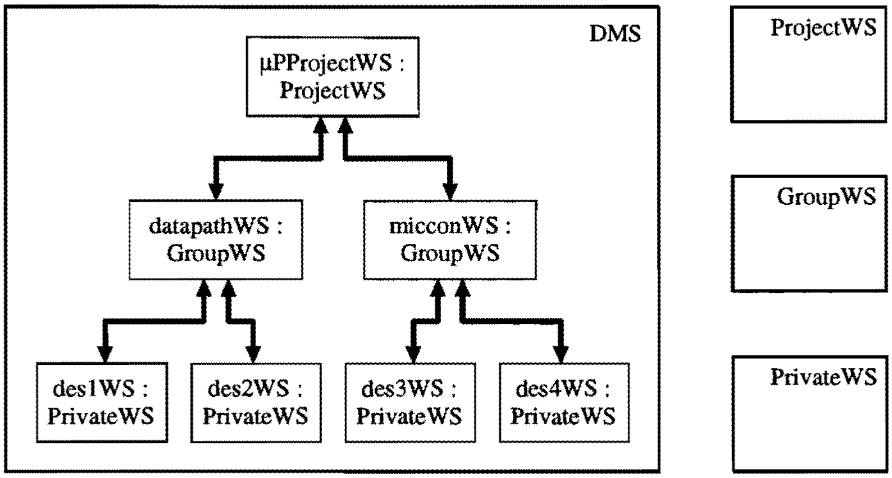


Figure 7.1: A decomposed design management system.

of seven communicating workspace subsystems: a project workspace "μPProjectWS", two group workspaces "datapathWS" and "micconWS" and four private workspaces "des1WS", "des2WS", "des3WS" and "des4WS". The "DMS" module does not contain the actual descriptions of these workspaces. Instead it defines these by referring to the module of which these are an instantiation. For example, "datapathWS" and "micconWS" are defined by stating that these are instantiations of the "GroupWS" module, which is a description of a group workspace. Hierarchical languages usually have a substitution semantics, i.e., each hierarchical design can be converted into an equivalent non-hierarchical design just by replacing the instantiations with copies of the corresponding module. This process of hierarchy elimination is usually referred as *flattening*.

In addition to their support for decomposition, hierarchical languages have a number of other advantages. Among others, hierarchical languages greatly promote design *reusability*. If a designer creates a useful module, then it can be reused in a number of other designs just by instantiating it. For example, it is very easy to construct a DMS for an arbitrary workspace hierarchy using the "ProjectWS", "GroupWS" and "PrivateWS" modules of Figure 7.1. To reuse a module, a designer does not need to have detailed knowledge about how the module is implemented. It suffices to know how to interact with the module interface in order to obtain the required functionality. So the module interface can be used to *abstract* from information about the internal structure of the module. In addition to abstraction, the module interface will also *encapsulate* the module contents, i.e., the module can only be connected with the rest of the design via the interface. This prevents illegal new connections from being made.

Our last requirement is: *The language should enable us to create design management system representations according to the DMS model presented in the previous chapters.* This is a rather complex requirement. However, if the design management system is structured as depicted in Figure 3.4, then this requirement can be replaced by a much simpler one. In Figure 3.4 the DMS is described in terms of a design management strategy independent core. This core is programmed by the DMS developers to perform its design management system services by downloading DMS representations. If the DMS is structured like this, then a DMS description language will satisfy the last requirement if these DMS representations can be expressed in it. In this case the DMS core can be implemented using a Petri net simulator executing these representations. So if the DMS is structured as shown in Figure 3.4, then the last requirement is reduced to: *The language should enable us to express the design management system representations used to program the DMS core.*

In the next section, which introduces interactive hierarchical coloured Petri nets, we demonstrate that languages based on this type of Petri net satisfy the first four requirements. In the last section of this chapter, we demonstrate that interactive hierarchical coloured Petri net based languages also satisfy our last requirement. We achieve this by showing how such a language can be used for design management system representation.

7.2 Interactive hierarchical coloured Petri nets

In this section we give a stepwise introduction into hierarchical coloured Petri nets [Jen92]. We first describe the simplest form of Petri net referred to as a place/transition net. Subsequently we show how the expressiveness of place/transition nets can be increased by the introduction of Petri net inscriptions. We follow this by a demonstration of how the abstraction level of Petri nets can be raised by the introduction of colours. Finally we show how the Petri net size can be reduced by the introduction of hierarchy.

Hierarchical coloured Petri nets lack support for describing user and operating system interactions and do therefore not satisfy the third requirement. This problem is solved by the introduction of interactive Petri nets, which feature special constructs to describe these interactions. In the last part of this section we will define this extension of the hierarchical coloured Petri net concept.

7.2.1 Place/transition nets

Petri nets have a graphical representation and have, in contrast to many other graphical description languages, a well-defined operational semantics. As a result of their operational semantics all Petri nets are executable, so Petri net based languages automatically satisfy our first requirement.

The simplest type of Petri net is a Place/Transition Net (PTN). Like its name suggests, the main components of such a net are places and transitions. These components are connected using arcs. As an example consider the Petri net depicted in Figure 7.2, which can be used to determine the minimum "c" of two numbers "a" and "b".

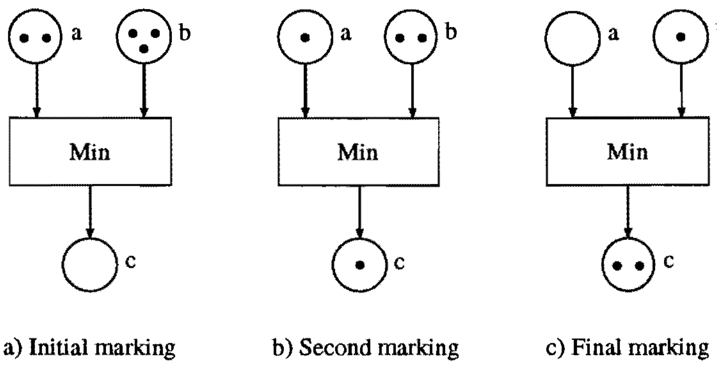


Figure 7.2: A Petri net which determines the minimum of two numbers.

Places are stores which contain the data being processed by the Petri net. Places, like "a", "b" and "c" in Figure 7.2, are usually depicted using a circle. *Transitions* determine what, given the current state of the places, the next state will be. Transitions, e.g., "Min" of Figure 7.2, are usually depicted using a rectangle. Transitions and places are connected by *arcs*. A transition will only influence the contents of those places it is connected to by arcs. If an arc points from a place to a transition, then the place is said to be an input of the transition. So "a" and "b" are the input places of transition "Min". Likewise, an arc pointing from a transition to a place, indicates that the place is an output of the transition. As an example consider the output place "c" of transition "Min".

The data being processed by a Petri net are organised in the form of *tokens*. The presence of a token in a place is indicated by putting a black dot in the corresponding circle. For place/transition nets data are represented by the number of tokens in a place. For example, the presence of two and three tokens in the places "a" and "b" of Figure 7.2.a, respectively, represents that 2 and 3 are the numbers of which the minimum is to be determined. An arbitrary distribution of tokens on the places is called a *marking*.

The marking of a Petri net is changed by the execution of transitions. Transitions can only *occur* when they are enabled. A transition is said to be *enabled* when every of its input places contains at least one token. Execution of a transition will consume one token from each of its input places and will add one token to each of its output places.

As an example of how a Petri net executes again consider the minimum calculator depicted in Figure 7.2. Figure 7.2.a shows the initial distribution of tokens on the places. This is called the *initial marking*. This initial marking represents that we want to determine the minimum of 2 and 3. Since both its input places "a" and "b" contain at least one token, the transition "Min" is enabled. Execution of "Min" will lead to the new marking depicted in Figure 7.2.b. For this new marking, "Min" is still enabled. A second execution of "Min" will lead to the final marking, in which no transitions are enabled anymore. In the final marking the place "c" contains two tokens, which represents that the result of the minimum calculation is 2.

During the execution of a Petri net it will often occur that a number of transitions are enabled simultaneously. From these enabled transitions an arbitrary number are selected and are then executed concurrently. The fact whether this form of concurrency is suitable for the representation of multi-user systems, so whether it satisfies the second requirement, strongly depends on how user interaction is added to the Petri net concept. Therefore, it will not be discussed here but in the section about interactive Petri nets.

7.2.2 Net inscriptions

Although it is possible to implement a wide variety of functions using the place/transition Petri nets described above, the representations of these functions will in general be very large and difficult to construct. For example, it is easy to implement a simple function like a minimum calculator, but what about the much more complex multiplication? However, PTNs can be made more expressive by the introduction of net inscriptions. We will now describe four types of net inscriptions: arc expressions, transition guards, place initialisers and transition expressions.

One of the reasons for lack of expressiveness of PTNs is that transitions are only able to perform a very simple transformation, i.e., remove one token from each of its input places and add one token to each of the output places. A possible remedy is to label the arcs of a Petri net with *arc expressions*. For an input arc, i.e., an arc connecting an input place to a transition, the corresponding arc expression will evaluate to the number of tokens the transition will consume from the input place when it is executed. Likewise, output arc expressions, i.e., expressions labelling an arc connecting a transition to one of its output places, evaluate to the number of tokens to be added to the corresponding output place if the transition occurs. Additional flexibility is added by allowing arc expressions which depend on the number of tokens in the input places. A transition with labelled arcs is enabled if its input places each contain at least the number of tokens specified by the corresponding arc expression. As an example consider the Petri net depicted in Figure 7.3, which implements a natural number multiplier. It operates by "b" times adding "a" to "c". Each execution of the addition transition "Plus" will remove one token from input place "b" and no tokens from input place "a". Therefore, this transition will be enabled as long as "b" contains tokens. The output arc expression "a" represents that each execution of "Plus" will add a number of tokens to "c" equal to the number of tokens currently in "a". To illustrate how the multiplier Petri operates consider the three markings depicted in Figure 7.3 corresponding to the multiplication of the naturals 3 and 2.

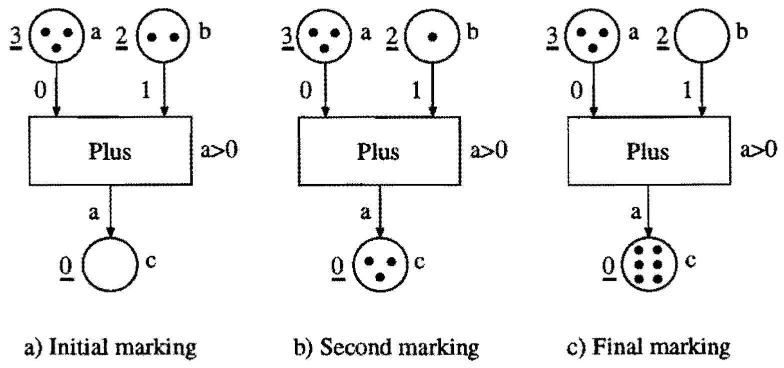


Figure 7.3: A multiplication Petri net.

Although arc expressions enable us to vary the number of tokens consumed/produced by a transition, the transition enabling rule has not changed; a transition is enabled when the required number of tokens are available at its input places. Sometimes this simple rule is inconvenient. For example, consider a multiplication using the Petri net of Figure 7.3, where "a" and "b" contain 0 and 1000 tokens, respectively. Since "Plus" is enabled as long as "b" contains tokens, "Plus" will transition a 1000 times only to find out that the result is still zero. This in contrast to the situation in which "b" is 0. A possible way to solve this is to extend the network in such a way that "b" will always contain the minimum of the numbers to be multiplied. This new network, however, will be much more complex than the original one. Another possibility is to extend the enabling rule by the introduction of Boolean functions referred to as *transition guards*. When a transition is labelled with such a guard, then this transition will be disabled as long as this guard evaluates to "false". When the guard evaluates to "true", then the old enabling rule will hold. As an example again consider Figure 7.3, where the transition "Plus" has been labelled with the guard "$a > 0$". This guard will disable this transition when "a" equals 0, thereby creating the desired effect.

As stated before, the execution of a Petri net starts from an initial marking. But how is this marking specified? One way to do this is by labelling the places of a Petri net with a *place initializer*. A place initializer is an expression which evaluates to the number of tokens the place will contain in the initial marking. As an example again consider Figure 7.3. The places "a", "b" and "c" have been labelled with the expressions "3", "2" and "0", respectively, which equal the number of tokens present in the initial marking depicted in Figure 7.3.a. Note that place initialisers are distinguished from the other place attributes by the fact that they are underlined.

As illustrated by the output arc expression "a" and the guard "$a > 0$" of the multiplication

Petri net of Figure 7.3, arc expressions and guards can be defined in terms of the contents of the input places. In addition to this, these net inscriptions may depend on the contents of the output places. They refer to the number of tokens currently stored in an input or output place using its name. In addition to this, we will allow the results produced by an arc expression to be referred to in the guard and the other arc expressions of the corresponding transition. We enable guards and arc expressions to access the results of these expressions by the introduction of a variable "I(p)" for each input arc and a variable "O(p)" for each output arc. The variable "I(p)" can be used to refer to the result of the expression labelling the arc connected to input place p. Likewise, "O(p)" can be used to access the result of the expression associated with the arc connected to output place p.

As an example of how arc expression variables are used consider the Petri net shown in Figure 7.4, which represents an order processing system used by a car manufacturer. The tokens stored in the "orders" input place model the number of currently available car orders. In a negotiation process of which the details are not modelled by our Petri net, the car manufacturer obtains a certain portion of these orders. This is represented by the arc expression "RS(orders)", where the "RS" command is used to Randomly Select a number between zero and "orders". The result of the evaluation of this expression can be referred in other expressions using "I(orders)". As an example consider the expression associated with the arc connected to the "capacity" input place. The "capacity" place models the available resources of the car manufacturer. If the capacity of the car manufacturer exceeds the number of orders obtained, so if "capacity $\geq$ I(orders)", then all orders will be processed and the available capacity will decreased with "I(orders)". In case the available resources are not sufficient, the complete capacity is used to process as much orders as possible. The result of the evaluation of the capacity arc expression, referred to using "I(capacity)", represents the number of accepted orders, i.e., those obtained orders for which sufficient capacity was available. In case the number of obtained orders exceeds the capacity, the remaining orders will be cancelled. This is represented by the output arc labelled with "I(orders) - I(capacity)", via which the cancelled orders are returned to the "orders" place. The car manufacturer produces its cars using two factories "fac1" and "fac2". Via a process of which the details are not modelled by our Petri, it is decided how many of accepted orders will be produced by factory "fac1". This is modelled by an output arc expression given by "RS(I(capacity))" for output place "fac1". The remaining accepted orders, of which the number is given by "I(capacity) - O(fac1)", are send to factory "fac2".

The use of a separate expression for each arc is inconvenient in situations where the arc expressions have much in common. To illustrate this consider the "difference" transition depicted in Figure 7.5. This transition determines the difference between the number of tokens currently stored in its input places "in1" and "in2" by subtracting the number of tokens in "in2" from the number of tokens stored in "in1" and by subsequently taking the absolute value. If the result of this operation is smaller than 8, then it is presented to the environment in binary form. This binary representation is stored in the output places "o2", "o1" and "o0", where "o2" contains the most significant and "o0" the least significant bit. The output arc expressions of the "difference" transition are almost identical. They

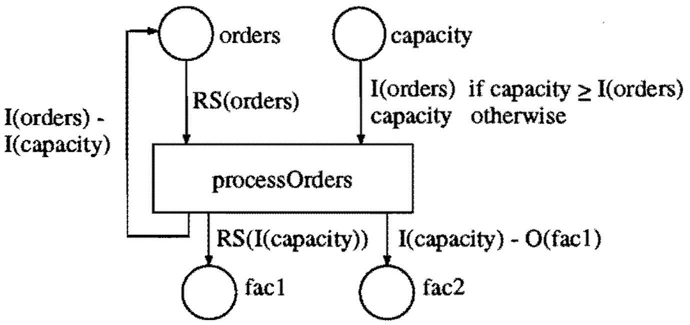


Figure 7.4: The car order processing system Petri net.

all contain the difference calculation and the conversion of the resulting difference from decimal to binary format as performed by the "dec2bin" function. They only differ in which bit they select from the resulting binary number. Since arc expressions are not able to share common subexpressions like "dec2bin(| in1 - in2 |)", these subexpressions will have to be duplicated. As a consequence the arc expressions can be much larger than necessary, resulting in transition descriptions which are more difficult to understand and time consuming to execute. We tackle this problem by the introduction of a limited form of code sharing for arc expressions based on transition expressions.

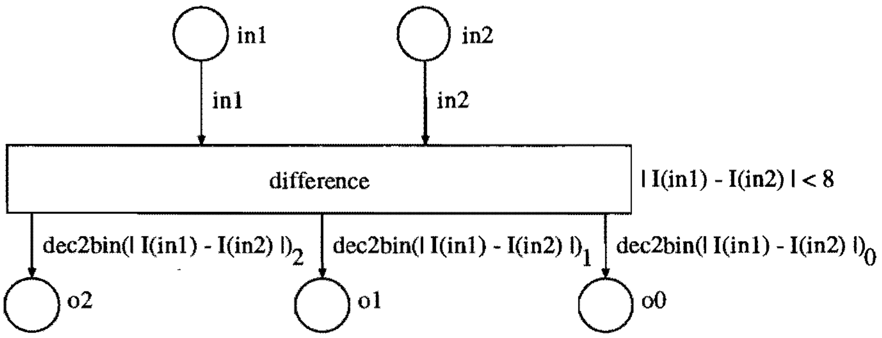

in1
in2
in1
in2
difference
| I(in1) - I(in2) | < 8
dec2bin(| I(in1) - I(in2) |)2
dec2bin(| I(in1) - I(in2) |)1
dec2bin(| I(in1) - I(in2) |)0
o2
o1
o0

Figure 7.5: The "difference" transition.

A *transition expression* is an expression representing those aspects of a transition which are not specific to a certain in- or output. The fact that a transition with guard g has a transition expression TE is indicated by labelling it with $g : TE$. In addition to transition expressions, we also introduce a variable "TE", which can be used in arc expressions to refer to the result of the transition expression. As an example of how the introduction of transition expressions can result in a reduction of the arc expression size, we refer to the much simpler description of the "difference" transition shown in Figure 7.6.

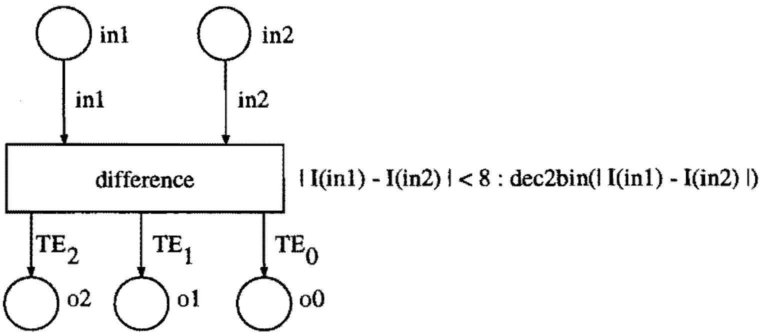


Figure 7.6: The simplified "difference" transition.

7.2.3 Coloured Petri nets

Although the introduction of net inscriptions greatly reduces the size of a Petri net, they do not change the fact that these Petri nets are descriptions at a very low level of abstraction. The abstraction level is comparable to that of a bit level digital system description. The only difference is that these Petri nets use the token number rather than a bit vector to represent data. The representation of a natural number multiplier is easily done at this abstraction level, but what about a much more complex operation like the multiplication of two real numbers. Although it is possible to represent such complex operations using the Petri nets introduced before, designing it will be a complex task and will result in a large net.

The abstraction level of Petri nets can be raised by the introduction of colours, resulting in a new type of net referred to as a *Coloured Petri Net* (CPN). Coloured Petri nets equip each token with a data value, called the *token colour*. The data value may be of arbitrary type, e.g. a real, a string or a record. Coloured Petri nets feature typed places; each place has a net inscription associated with it referred to as its *colour set*, which represents a data type. A token can only be added to a place if its token colour belongs to the corresponding colour set. The fact that a place p has type T is indicated by labelling this place with $p : T$. So if the colour set of a place "a" is equal to the set of reals $\mathbb{R}$, then this place will be labelled with "a:$\mathbb{R}$".

One of the consequences of the introduction of token colours is that the contents of a place can be described no longer by only giving the number of tokens it contains. Instead, we have to specify which colours occur and the number of tokens for each of these colours. As an example consider a place with a colour set equal to the set of characters Char. If the place contains six tokens with the colours "x"', "x", "y", "y", "y" and "z", respectively, then the contents of this place can be characterised by stating that it contains 2 tokens with colour "x", 3 tokens with colour "y" and 1 token with colour "z". [Jen92] introduces multi-sets to represent this kind of information. In contrast to sets, multi-sets are allowed to contain multiple occurrences of an element. Multi-sets can be represented by a set of pairs associating each of its elements with an occurrence count. The multi-set "ms" representing the contents of the place of our example is given by $\{<2, \mathrm{x}>, <3, \mathrm{y}>, <1, \mathrm{z}>\}$. In the rest of this chapter we will use the more succinct *formal sum* notation to represent multi-sets. This notation combines the multi-set pairs using the + symbol and omits the symbols combining the elements of a pair. Using this notation the multi-set of our example is given by 2'x + 3'y + 1'z. For reasoning about multi-sets it is often more convenient to represent a multi-set by a function which maps each of the multi-set elements to its number of occurrences. The multi-set of our example can be represented by a function ms : Char $\rightarrow \mathbb{N}$ defined by

$$\mathrm{ms}(s) = \begin{cases} 2 & \text{if } s = \mathrm{x} \\ 3 & \text{if } s = \mathrm{y} \\ 1 & \text{if } s = \mathrm{z} \\ 0 & \text{otherwise} \end{cases} .$$

In [Jen92] the following formal multi-set definition is found.

Definition 7.2.1 *Multi-set*

A multi-set m, over a non-empty set S, is a function $m : S \rightarrow \mathbb{N}$.
The non-negative integer $m(s) \in \mathbb{N}$ is the number of appearances of the element s in the multi-set m. For a multi-set function m the corresponding formal sum is given by

$$\sum_{s \in S} m(s)\text{'}s.$$

For a set S the set of all multi-sets over S is denoted by S_{MS}.

The empty multi-set ϕ is a multi-set containing no elements. When defined over a set S, it is defined by $\phi(s) = 0$ for all $s \in S$.

In [Jen92] a number of operations on multi-sets are defined. In this thesis we make use of the *addition* operator +, the *subtraction* operator - and the *smaller than or equal* relational operator $\leq$. The results of the application of these operators to two multi-sets m_1 and m_2 are given by

$$m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))\text{'}s, \tag{7.1}$$

$$m_2 - m_1 = \sum_{s \in S} (m_2(s) - m_1(s))\text{'}s, \text{ and} \tag{7.2}$$

$$m_1 \leq m_2 = \forall s \in S : m_1(s) \leq m_2(s). \tag{7.3}$$

Note that the subtraction operation can only be performed when $m_1 \leq m_2$. If $m_1 \leq m_2$ holds, we say that m_1 is a *subset* of m_2. In addition to the operators defined above we also introduce the *element-of* operator $\in$. For a multi-set m over a set S, we define this operator as follows

$$s \in m = s \in S \wedge m(s) > 0. \tag{7.4}$$

Just as for a PTN, the places of a CPN are initialised using a place initializer. However, for a CPN this function will not produce a number but a multi-set to represent the initial contents of the corresponding place. To illustrate this consider the real number multiplier of Figure 7.7. Evaluation of the initializer of place "a" will result in the multi-set 1'2 + 1'5, which denotes that after initialisation place "a" will contain two tokens with colour 2 and 5, respectively.

As we have seen before, a transition operates by removing tokens from its input places and adding tokens to its output places. For coloured Petri nets it no longer suffices to specify the number of tokens involved. Instead, a multi-set is required to indicate which tokens are to be removed from or to be added to a place. Therefore, the arc expressions of a CPN will generate a multi-set rather than a natural number. As an example consider the real number multiplier CPN of Figure 7.7. The transition "Mult" takes an arbitrary token from each of its input places "a" and "b", multiplies the corresponding token colours and when the result is smaller than 16, it subsequently adds a token labelled with this value to its output place "c". The input arc expressions select the values to be multiplied from the corresponding input place multi-set using the "RC" command. When applied to a multi-set, "RC" will Randomly select one of the Colours occurring in this set. The result of the evaluation of the input arc expressions is a multi-set containing only one element, namely the selected value. The guard and the output arc expression use the "RC" command to select the values to be multiplied from these sets. When applied to a multi-set consisting of a single element, "RC" will return the element involved. The result of the evaluation of the output arc expression is a multi-set containing the result of the multiplication as its only element.

For the examples presented in the rest of this chapter, most arc expressions will generate a multi-set containing exactly one token. To simplify our notation we will replace such an expression by a simpler arc expression, which generates the token value rather than the multi-set containing it. As an example consider the input arc expressions of the "Mult" transition of Figure 7.7, which can now be reduced to "RC(a)" and "RC(b)". An additional advantage is that other arc expressions referring to the result of such an expression can also be reduced, because they do not have to select the token anymore. As a result of these simplifications the output arc expression of the "Mult" transition is reduced to "I(a)*I(b)", resulting in the much better readable CPN description of the real number multiplier depicted in Figure 7.8.

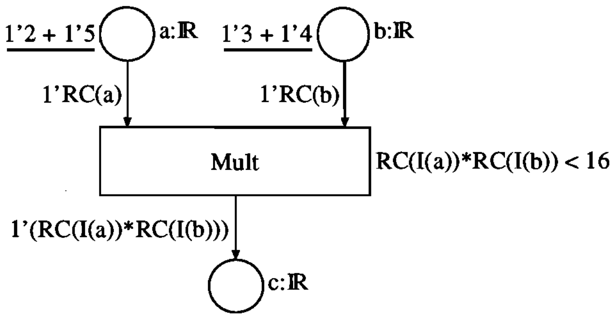


Figure 7.7: A real number multiplier CPN.

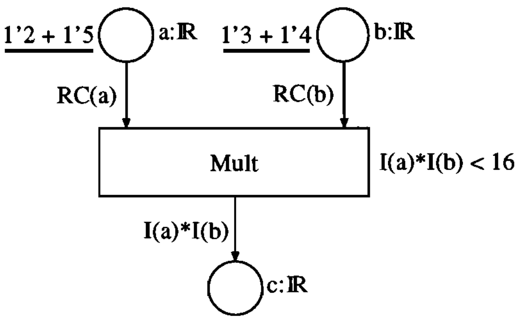


Figure 7.8: A simplified real number multiplier CPN.

7.2.3.1 Execution of coloured Petri nets

All the types of Petri nets we have introduced are executable, so languages based on these Petri nets automatically satisfy our first requirement. Descriptions created using such a language are executed using a Petri net simulator. The Petri net semantics leaves us a lot of freedom in choosing the execution method to be used by the simulator: A variety of methods can be devised differing in how efficiently they execute the Petri net. In this section we will describe an execution method developed for reasons of clarity rather than efficiency. In the rest of this chapter we will refer to this method as the *simple execution method.*

In the first phase of the execution process, coloured Petri net simulators determine the initial marking. To obtain this marking the simple execution method performs the following steps:

1. Evaluate the place initialisers.
2. Add the tokens specified by the resulting multi-sets to the corresponding places.

Starting from a given marking, the simple method determines the next marking as follows:

1. Randomly select the transitions to be evaluated.
2. Evaluate the selected transitions.
3. Make a random selection from the set of evaluated transitions, such that the selected transitions are concurrently enabled.
4. Execute the transitions selected in the previous step concurrently.

To illustrate the simple Petri net execution method, we again use the real number multiplier CPN of Figure 7.7. Evaluation of its place initialisers results in the multi-sets 1'2 + 1'5 and 1'3 + 1'4 for the places "a" and "b", respectively. As a result of this "a" is initialised by the addition of two tokens coloured 2 and 5. Likewise, two tokens with values equal to 3 and 4 are added to "b".

From a given marking, the next marking is obtained by performing the four steps described above. In the first step the transitions to be evaluated are randomly selected. Since "Mult" is the only transition of our Petri net, it is the only candidate for being evaluated and subsequently executed. Therefore, the choice seems not to be difficult. However, Petri net transitions can be executed concurrently with themselves. For example, starting from the initial marking described above, the next marking can be obtained by executing "Mult" two times, once for "I(a)" and "I(b)" equal to 1'2 and 1'4 and once for the combination 1'5 and 1'3. As a result of this concurrent execution of "Mult" with itself a new marking is obtained featuring empty "a" and "b" input places and a "c" output place containing the resulting values 8 and 15. Concurrent execution of a transition with itself using different input arc expression values is only possible if the transition has been evaluated multiple times. For our example, this means that "Mult" its arc expressions and guard must have been evaluated at least two times. So when selecting the Petri net transitions to be evaluated, we not only have to specify which transitions are involved but also how many times their evaluation has to be done. This information can be represented by a multi-set over the set of transitions. So if "Mult" is selected to be evaluated four times, then the result of step 1 will be given by the multi-set 4'Mult.

During the evaluation of a transition the values associated with its arc expressions, its guard and transition expression are determined. In case of the simple execution method, net inscriptions are evaluated sequentially. We will represent the result of a transition evaluation in the form of a *transition binding.* A binding *binds* net inscriptions to the result of their evaluation. Bindings can represented by a set of $< variable, value >$ pairs, where *variable* the variable used to refer to a net inscription and *value* the associated result of its evaluation. As an example consider the following binding of the "Mult" transition.

{<I(a),1'2>,<I(b),1'3>,<G,true>,<O(c),1'6>}. (7.5)

Besides the input arc expression variables "I(a)" and "I(b)", which are bound to the 1'2 and 1'3 multi-sets and the output arc expression variable "O(c)" which is bound to 1'6, this binding also associates the value "true" with variable "G", which is used to refer to the value of the guard.

During the transition evaluation step all the selected transitions are evaluated as specified by the multi-set produced by step 1. The result of the evaluation step can be represented by a multi-set, referred to as the *evaluation multi-set*, which contains all the evaluations performed. We represent an *evaluation* by a pair $< trans, bind >$, where *trans* a transition and *bind* its binding. To illustrate this again consider the multiplier of Figure 7.7. Since the result of step 1 was given by the multi-set 4'Mult, "Mult" will have to be evaluated four times during the evaluation step. The result of these four evaluations can be represented by an evaluation set like

$$1' < \text{Mult}, B_1 > +1' < \text{Mult}, B_2 > +1' < \text{Mult}, B_3 > +1' < \text{Mult}, B_4 >, \quad (7.6)$$

where the bindings B_1, B_2, B_3 and B_4 are given by

B_1 = {<I(a),1'2>,<I(b),1'3>,<G,true>,<O(c),1'6>},
B_2 = {<I(a),1'2>,<I(b),1'4>,<G,true>,<O(c),1'8>},
B_3 = {<I(a),1'5>,<I(b),1'3>,<G,true>,<O(c),1'15>}, and
B_4 = {<I(a),1'5>,<I(b),1'4>,<G,false>,<O(c),1'20>}.

During the third step a selection is made from the evaluation set, such that the selected transitions are concurrently enabled with respect to the associated bindings. The result of this step can be represented by a subset of the evaluation set. We will refer to this set as the *step set*, because it describes the *step* taken by the simulator to obtain the next marking from the given marking.

In order for a transition to be concurrently enabled with other transitions it first has to be enabled itself. A transition is said to be enabled with respect to a binding if and only if the guard specified by this binding equals "true" and the input places of the transition contain the tokens specified by the input arc expression variables. So the transition "Mult" of our example is not enabled with respect to the binding B_4, because the guard variable G equals "false". The guards associated with the other bindings B_1, B_2 and B_3 are "true". Moreover, for the initial marking of our multiplier Petri net the input places of "Mult" contain the tokens specified by these bindings. For example, binding $B1$ specifies that input place "a" has to contain a token with colour 2 and that place "b" has to contain a token with value 3, which happen to be available at these places. Therefore, "Mult" is enabled with respect to the bindings B_1, B_2 and B_3. A transition which is enabled for a certain binding can be executed, thereby consuming and producing tokens according to the arc expression values specified by the binding.

The transitions associated with the step set have to be concurrently enabled, i.e., their input places have to contain sufficient tokens to concurrently execute these transitions for each of the corresponding bindings. To illustrate this consider the step set given by

$$\{<\text{Mult}, B_2>, <\text{Mult}, B_3>\}, \tag{7.7}$$

where B_2 and B_3 are the bindings as defined by Equation 7.6. In our initial marking "Mult" is enabled for both binding B_2 and binding B_3. Transition "Mult" is also concurrently enabled with respect to these bindings, because the input tokens required for the concurrent execution of "Mult" for each of these, i.e., 2 and 5 from "a" and 3 and 4 from "b", are available at these input places. As an example of a subset of the evaluation set which is not a valid step consider the set

$$\{<\text{Mult}, B_1>, <\text{Mult}, B_2>\}, \tag{7.8}$$

where the bindings B_1 and B_2 again as specified by Equation 7.6. Although the "Mult" transition is enabled with respect to both B_1 and B_2 individually, it is not concurrently enabled with respect to these bindings, because input place "a" does not contain the required two tokens with value 2.

During the execution step the next marking is obtained by concurrent execution of the transitions specified by the step set produced during step 3. In case of our example, concurrent execution of the "Mult" transition for the bindings specified by the step set of Equation 7.7 will result in the removal of all the tokens from the input places "a" and "b" and the addition of two tokens with values 8 and 15 to output place "c".

7.2.3.2 Coloured Petri net behaviour

Coloured Petri nets behave in a non-deterministic way, i.e., starting from a given marking a number of next markings are possible. To illustrate this again consider our multiplier Petri net. In the previous section we have shown that the next marking can be obtained by executing the "Mult" transition twice, namely once for binding B_2 and once for binding B_3. However, in addition to this, there are three alternative next markings, which are the result of a single execution of the "Mult" transition using a binding given by either B_1, B_2 or B_3. For coloured Petri nets there is nothing specified about which of these markings should be chosen. The actual choice is determined by the details of the execution method used by the simulator. As a result of this, there is wide range of possible execution methods, which differ in the choices they make in the case of non-determinism.

As we will demonstrate later on in this chapter, the Petri net execution method described in the previous section is not very suitable for the simulation of interactive Petri nets. We are going to solve this problem by defining a new execution method. For defining such a new method it is essential to have a good understanding of the Petri net semantics. In the rest of this section we give a formal definition of coloured Petri net behaviour. This treatment was inspired by the formalisation found in [Jen92].

Before we can start our discussion about coloured Petri net behaviour, we first have to know what a coloured Petri net is. We define coloured Petri nets as follows.

Definition 7.2.2 *Coloured Petri net*

A *coloured Petri net* is a 7-tuple $< \sum, P, T, A, \mathit{Colour}, \mathit{Init}, \mathit{TNI} >$ where

- $\sum$ a finite set of non-empty types, called *colour sets*,
- P a finite set of places,
- T a finite set of transitions,
- A a finite set of arcs,
- *Colour* the colour function,
- *Init* the initialisation function, and
- *TNI* the transition net inscription function.

Arcs connect places and transitions. An input arc, i.e., an arc leading from an input place p to a transition t, can be represented by a pair $< p, t >$. Likewise, an output arc can be represented by a pair $< t, p >$, denoting that it connects a transition t to an output place p. As a result of this the following will hold for the arc set A.

$$A \subset P \times T \cup T \times P. \tag{7.9}$$

The *colour function Colour* is a function defined from P into $\sum$. It maps each place p onto a colour set $\mathit{Colour}(p)$. Each token stored in place p will have a token colour belonging to type $\mathit{Colour}(p)$.

The *initialisation function Init* maps each place p onto a multi-set over $\mathit{Colour}(p)$. This multi-set describes which tokens will be added to p during the initialisation step.

The *transition net inscription function TNI* is a function which, when applied to a transition t and a *transition variable* v, returns a *net inscription function* ni. A transition variable is a variable used to refer to one of the net inscriptions of a transition. A transition variable v of a transition t is a member of the *transition variable set* $\mathit{Var}(t)$, which contains all the variables introduced to refer to the results produced by the input and output arc expressions, the transition expression and the guard. Therefore, this set is defined by

$$\mathit{Var}(t) = \{\mathrm{I}(p) \mid < p, t > \in A\} \cup \{\mathrm{O}(p) \mid < t, p > \in A\} \cup \{\mathrm{TE}, \mathrm{G}\}\ . \tag{7.10}$$

A net inscription function ni is a function which, based on the current contents of the input and output places of the corresponding transition and the results associated with some of the net inscriptions, produces a value describing the results of the evaluation of the net inscription it represents. The variables on whose value the result produced by a net inscription function ni of a transition t depends are listed in the corresponding *net inscription variable set* $\mathit{Var}(ni, t)$. For this set the following holds

$$Var(ni,t) \subset Var(t) \cup \{p| < p,t > \in A \vee < t,p > \in A\}. \tag{7.11}$$

Having defined what a coloured Petri net is, we can now start our formalisation of coloured Petri net behaviour. Petri nets behave by adding and removing tokens to and from their places, i.e., by changing their marking. Coloured Petri net markings are defined as follows.

Definition 7.2.3 *Marking*

A marking is a function M defined on the set of places P such that

$\forall p \in P : M(p) \in Colour(p)_{MS}$.

For a place p $M(p)$ returns a multi-set on $Colour(p)$ representing the current contents of this place.

The *initial marking* M_0 is the marking which is obtained by evaluating the initialisation expressions. Therefore, it is defined by

$$\forall p \in P : \mathrm{M}_0(p) = Init(p)\ . \tag{7.12}$$

From a given marking the next marking is obtained by execution of a number of transitions. Which transitions can be executed and how this affects the corresponding places is determined by the associated net inscriptions. The values associated with these net inscriptions can be calculated by application of the corresponding net inscription function using the values currently associated with the corresponding net inscription variables. We represent the information about the values associated with these variables in the form of a *net inscription binding*. After evaluation of the net inscriptions of a transition, all the corresponding transition variables will be bound to the resulting values. We represent this information in the form of a *transition binding*. These two types of bindings are formalised as follows.

Definition 7.2.4 *Bindings*

A *transition binding* of a transition t is a function b on $Var(t)$ such that $b(v)$ returns the value associated with transition variable v. This value is given by

$b(v) = TNI(t,v)(nib)$,

where nib the net inscription binding associated with net inscription function $TNI(t,v)$.

A *net inscription binding* nib of a net inscription ni of a transition t is a function on $Var(ni,t)$ such that $nib(v)$ returns the value associated with net inscription variable v. In case of a current marking M and a binding b for transition t this value is given by

$$nib(v) = \begin{cases} M(v) & \text{if } v \in P \\ b(v) & \text{otherwise} \end{cases} .$$

We will represent bindings by a set of $< variable, value >$ pairs. As an example consider the transition binding of the "Mult" transition described by Equation 7.5. From now on we will use B to refer to the set of all possible transition bindings.

The current marking of a coloured Petri net is transformed into the next marking by the execution of what we refer to as a step. A step describes how this transformation is accomplished by stating which transitions have to be executed and for which bindings this has to be done. A step can be defined as follows.

Definition 7.2.5 *Step*

A step Y is a non-empty multi-set over $T \times B$. The fact that

$Y(< t, b >) = n$

denotes that transition t has to executed n times using binding b.

Not all steps can be executed. For some steps this is caused by the fact that the places do not contain enough tokens to concurrently execute the specified transitions for the associated bindings. A step can also not be executed if one of its bindings has a guard equal to false, which means that the corresponding transition can not be executed for this binding. A step which can be executed is referred to as an enabled step. We now formalise this using the following definition.

Definition 7.2.6 *Enabled step*

A step Y is *enabled* in a marking M if and only if the following property is satisfied.

$$\forall < t, b > \in Y : b(G) = true \wedge \forall p \in P : \sum_{<t,b> \in Y} b(\mathrm{I}(p)) \leq M(p) \ .$$

As an example of an enabled step consider the step set of Equation 7.7.

When a step is enabled it can occur. For a given marking there can be number of enabled steps, each of which produces another next marking. For a Petri net there is nothing specified about which of these steps will occur, i.e., the behaviour of a Petri net is non-deterministic. Which of the enabled steps is actually selected to be executed is determined by the details of the execution method used by the Petri net simulator. When a step occurs it will change the current marking as follows.

Definition 7.2.7 *Occurring step*

When a step Y is enabled in a marking M_1, it may *occur* changing the marking M_1 to another marking M_2, defined by

$$\forall p \in P : M_2(p) = (M_1(p) - \sum_{<t,b> \in Y} b(I(p))) + \sum_{<t,b> \in Y} b(O(p)).$$

As an example of how the occurrence of a step changes the current marking again consider the step specified by the step set of Equation 7.7. Execution of this step for the initial marking of the multiplier Petri net depicted in Figure 7.7 will result in the removal of all the tokens from the input places "a" and "b" and the addition of two tokens with values 8 and 15 to output place "c".

7.2.4 Hierarchical Petri nets

Hierarchical languages provide extensive support for decomposition and therefore satisfy our fourth requirement. There are a number of ways in which hierarchy can be introduced for coloured Petri nets [Jen92]. For now we restrict ourselves to the form of hierarchy obtained by the introduction of *composite transitions*. Unlike normal transitions, the behaviour of composite transitions is not described by arc expressions, but by a CPN network. As an example, consider the hierarchical CPN depicted in Figure 7.9 representing a 2-bit adder. This CPN consists of two modules; the full adder module "FullAdd" and the 2-bit adder module "Adder2". The module "Adder2" forms the top of the hierarchy. It contains two composite transitions: "fa_1" and "fa_2". Note that these composite transitions are indicated using a rectangle with a fat borderline. The transitions "fa_1" and "fa_2" are instantiations of the full adder module. This information is added to the coloured Petri net by typing the composite transition, i.e., the fact that a composite transition ht is an instantiation of module M is indicated by labelling it with $ht : M$. In our example this results in "fa_1" and "fa_2" being labelled with "fa_1:FullAdd" and "fa_2:FullAdd", respectively.

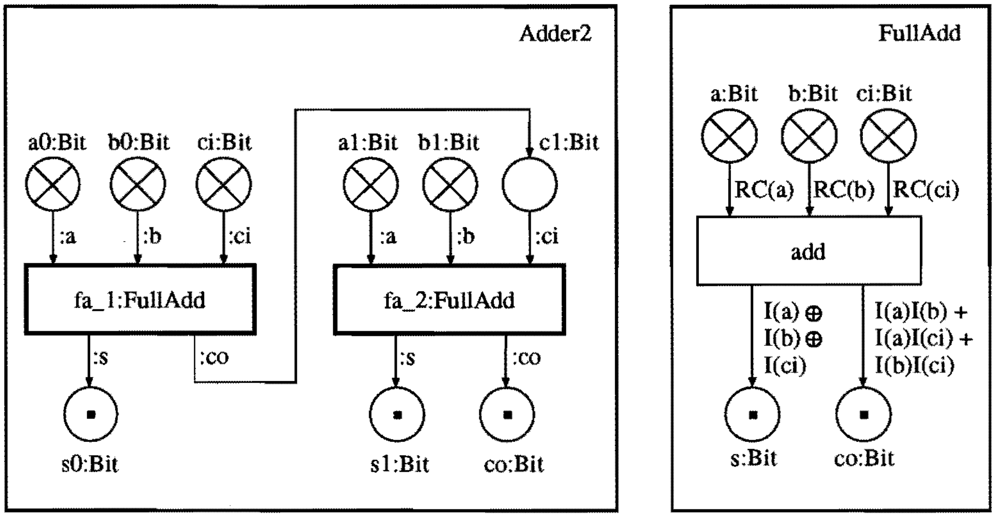


Figure 7.9: The "Adder2" hierarchical CPN.

What is the relation between the operation of a composite transition and the CPN of the corresponding module? The answer is that the behaviour of a CPN containing a certain composite transition is identical to that of a larger CPN, which is obtained by substitution of the composite transition and its arcs by a copy of the network of the corresponding module. The arc labels of a composite transition indicate how this substitution has to be done. If an arc is connected to a place P and labelled with $:MP$, then this informs us that place P has to be merged with place MP of the corresponding module. We will name the resulting place P. As an example consider the composite place "fa_1". The arc labels inform us that the places "a0", "b0", "ci", "s0" and "c1" have to be merged with the places "a", "b", "ci", "s" and "co" of module "FullAdd", respectively. Places are merged by replacing these by a new place which is connected to all the arcs of the original ones. As an example, consider the flattened CPN of Figure 7.10, which exhibits the same behaviour as the hierarchical CPN shown in Figure 7.9. Note that we have maintained transition name uniqueness by prefixing the name of the "add" transition with the name of the corresponding composite transition.

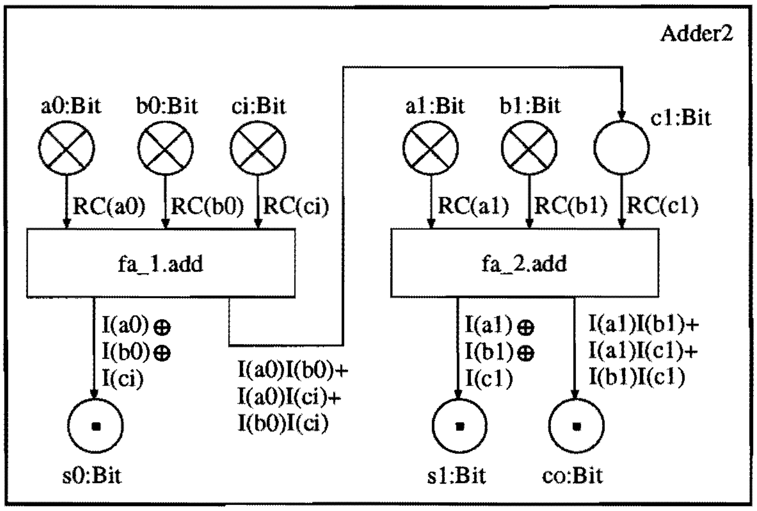


Figure 7.10: The "Adder2" flattened CPN.

The input and output places of a composite transition can only be merged with certain places of the corresponding module. An input place can only be combined with one of the module's input places, which are marked with a cross. Likewise, an output place can only be connected to an output place of the module. Module output places are distinguished from the other places by a black box marking their centre. As an example again consider the composite transition "fa_1". Its input places "a0", "b0" and "ci" can only be merged with the input places "a", "b" and "ci" of module "FullAdd". Likewise, the output places "s0" and "c1" can only be combined with either "s" or "co".

7.2.5 Interactive Petri nets

The Petri nets encountered until now lacked any form of user or operating system interaction. Therefore they do certainly not satisfy the third requirement. Petri nets can be made interactive by the introduction of interaction constructs which are to be used in the net inscription definitions. As an example consider the interactive multiplication Petri net shown in Figure 7.11. In contrast to the multiplier of Figure 7.8, the values to be multiplied are not randomly selected, but chosen from a menu. These choices are made by a user whose name is randomly selected from the contents of the "user" input place. The multiplication Petri net is made interactive using the interaction construct "menu(*User*, *Set*)", whose evaluation will pop-up a menu enabling user *User* to select an element of the multi-set *Set*. Evaluation of this construct will return the value selected by the user. The "menu" command is just an arbitrary example of a user interaction construct which can be added to the net inscription language of a Petri net. A real interactive Petri net based language will feature a collection of such interaction constructs, which either are used to implement user or operating system interaction.

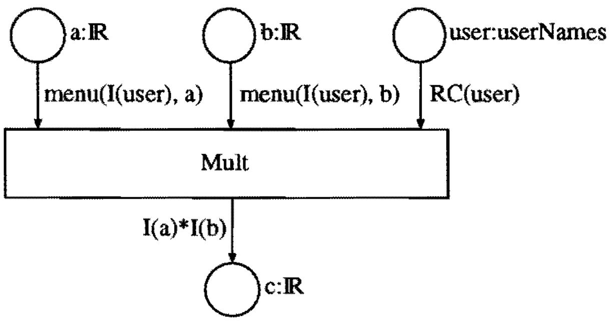


Figure 7.11: An interactive multiplication Petri net.

The addition of user interaction constructs to the Petri net net inscription language is a very simple extension, which in principle does not have to affect the way the Petri net is executed. In practice however, a number of serious problems are encountered when we attempt to use methods devised for normal coloured Petri nets for the execution of interactive Petri nets. To illustrate this we now use the interactive multiplication Petri net shown in Figure 7.11 to introduce one of these problems. Consider a situation where the "menu(I(user),a)" construct is evaluated for a certain user and the user involved happens to be off for a cup of coffee. In case the simple Petri net execution method described in Section 7.2.3.1 is used this will cause problems when the multiplication Petri net is part of a bigger Petri net servicing multiple users. The reason is that this method specifies that net inscriptions are evaluated sequentially. As a consequence, the evaluation of the interaction construct "menu(I(user),a)" will block the progress of the complete Petri net until the user involved has finished his cup of coffee. So although the introduction of interaction constructs results in a language satisfying our third requirement, it can also seriously reduce the amount of concurrency, resulting in a language which does not satisfy our second requirement.

In the next section Section 7.2.5.1 we will describe the problems encountered when the simple mechanism is used for the execution of interactive coloured Petri nets. This is followed by Section 7.2.5.2 in which a new Petri net execution method is introduced solving these problems.

7.2.5.1 Interactive CPNs and the simple execution mechanism

To illustrate the problems encountered when using the simple method for the execution of interactive coloured Petri nets, consider the interactive Petri net depicted in Figure 7.12. Execution of this Petri net will result in the invocation of the OMA tool.

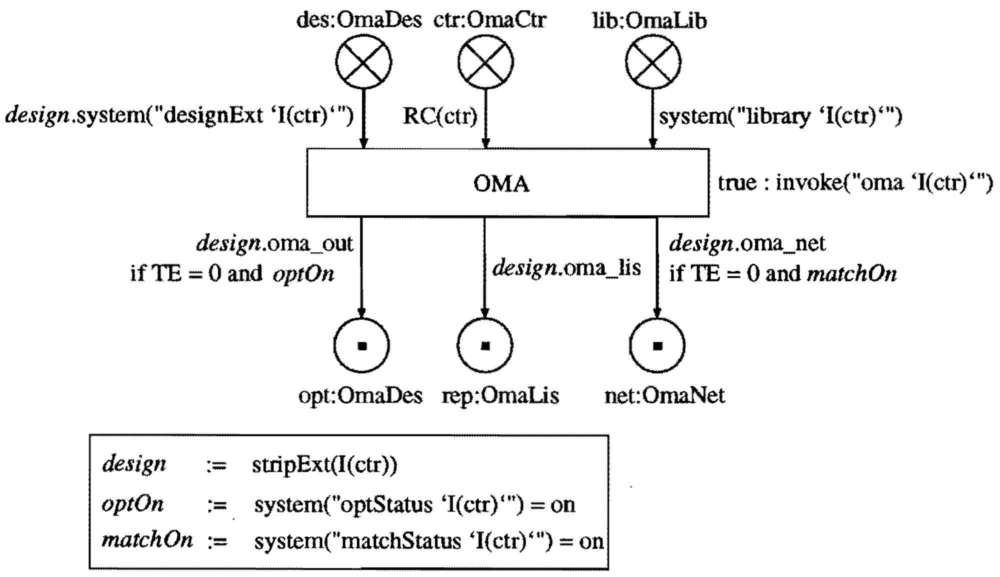

des:OmaDes
ctr:OmaCtr
lib:OmaLib
design.system("designExt 'I(ctr)'")
RC(ctr)
system("library 'I(ctr)'")
OMA
true : invoke("oma 'I(ctr)'")
design.oma_out
if TE = 0 and *optOn*
design.oma_lis
design.oma_net
if TE = 0 and *matchOn*
opt:OmaDes
rep:OmaLis
net:OmaNet
design := stripExt(I(ctr))
optOn := system("optStatus 'I(ctr)'") = on
matchOn := system("matchStatus 'I(ctr)'") = on

Figure 7.12: Interactive CPN representation of OMA.

OMA invocations are controlled by a control file. The names of the currently available control files are stored in the "ctr" input place. From this place the name of the control file to be used is randomly selected. Besides the control file, OMA has two additional inputs:

a design file and a library. The fact that a design file is available is represented by the presence of the corresponding name in the "des" input place. Likewise, the names of the available libraries are listed in the "lib" input place.

Which design file and library are used during an OMA run is uniquely determined by the control file. The name of the design file can be constructed by first stripping the "oma_ctr" extension from the control file name using the "stripExt" command, followed by an append of the design file type. The information about the design file type is stored in the control file. We now assume that there exists a small program "designExt", which when invoked on a control file extracts the design file type. If this program prints the result to standard output, then the design file type can be obtained using the interaction construct "system("designExt 'I(ctr)'")". Evaluation of the "system" command will order the operating system to execute the string specified by its argument and will return the output send to standard out as its result. In the rest of this chapter we will frequently encounter strings like "designExt 'I(ctr)'", i.e., strings of which certain parts have been surrounded by quotes. The quoted parts of such a string must not be taken literally, but as a reference to the result of their evaluation by the Petri net simulator. For example, if the selected control file name equals "controlFile.ctr", then the evaluation of the quoted expression "I(ctr)" will return "controlFile.ctr". As a result of this "designExt 'I(ctr)'" will refer to the string "designExt controlFile.ctr". In addition to "designExt", we assume that there exists a program called "library", which extracts the library name from the control file. If this program prints its result to standard output, then also the library name can be obtained using the "system" command.

Besides a guard which always evaluates to true, the "OMA" transition also features a transition expression given by "invoke("oma 'I(ctr)'")". Just like "system", the "invoke" command will order the operating system to execute the string specified by its argument. The result returned by this command however, is the resulting exit code rather than the information sent to standard out. Failure of an OMA run is indicated by an exit code unequal to zero. In this case it is not guaranteed that valid OMA design and netlist output files exist. For our "OMA" transition this is represented by output arc expressions which only produce the corresponding file name if the run was successful, which is indicated by a transition expression variable "TE" equal to zero. The listing file is always produced, because in case of run errors it will contain information about what went wrong.

Even in case of a successful OMA run the design and netlist files will not always be produced. When OMA is operated in the optimisation mode, it will not match the design and will therefore not produce the netlist output. Likewise, when run in the matching mode, OMA will perform no optimisations and will consequently not generate the optimised design output. The information about the execution mode of OMA is stored in the control input file. By default OMA will perform both optimisation and matching. Optimisation is switched off by adding the statement "optimising = off" to the control file. Likewise, matching is disabled using the control file statement "matching = off". So the content of the OMA control file determines whether or not execution of the "OMA" transition should add tokens to its "opt" and "net" output places. To obtain this information from the control file we introduce the programs "optStatus" and "matchStatus". When applied to a control file, "optStatus" and "matchStatus" produce either "on" or "off" indicating whether optimisation and matching will be enabled or disabled, respectively. The output arc expressions of the arcs connected to the "opt" and "net" places use the "system" command to run these commands. They use the result to determine whether or not a token has to be produced. Note that we have simplified the notation of these output arc expressions by omitting the "otherwise" case of the "if" statement. From now on we adopt the convention that, if the "otherwise" case is not explicitly specified in an conditional expression, it will default to the empty multi-set. For our output arc expressions this implies that in the otherwise case no tokens are to be produced for their output places during "OMA" its execution.

We have already discussed the first problem encountered when using the simple execution method for interactive Petri nets, namely that the evaluation of an interaction construct can block the progress of the complete Petri net. We will refer to this problem as the *blocking problem.* Like the interactive multiplication Petri net of Figure 7.11, evaluation of the "OMA" transition can result in a blocking problem. The reason is that the evaluation of its transition expression will result in the invocation of the OMA tool. As a result of this the evaluation of this expression will take as long as the time required for the OMA run, thereby blocking Petri net progress for a considerable amount of time.

At first glance, the blocking problem described above can be solved by switching from sequential to concurrent net inscription evaluation. This is however not sufficient. The reason is that the simple execution method prescribes that *all* the net inscriptions have to be evaluated before transitions can be executed. So despite the fact that concurrent evaluation enables us to evaluate the net inscriptions of the other transitions during the OMA run resulting from "OMA" its transition expression, none of these transitions can be executed until after this run has finished. A possible solution to this problem is to introduce a more flexible mechanism, in which a transition can be executed the moment it becomes enabled, independent from the status of the evaluation of other transitions.

The second problem encountered when applying the simple execution method to interactive coloured Petri nets is the one we refer to as the *unwanted evaluation problem.* Since the evaluation of interaction constructs can have side-effects and often uses a lot of resources, their evaluation should only be done if it is absolutely necessary. The simple execution method however, provides no support for controlling whether and when the evaluation of these constructs is done and will therefore often result in unwanted evaluations. To illustrate this, we are now going to describe the problems encountered when the simple method is used for the execution of the OMA interactive Petri net of Figure 7.12.

During the first step of the simple execution method it is determined which transitions are to be evaluated and how many times this has to be done. Suppose now that we have a current marking in which the "ctr" place contains a single token and that the "des" and "lib" places contain the tokens representing the corresponding design file and library, respectively. In this case the tokens stored in the input places only suffice to execute the "OMA" transition once. Therefore it is also not necessary to evaluate the "OMA" transition more than once. Despite this however, the outcome of step 1 can be that the "OMA" transition has to be evaluated four times. Since "ctr" contains only one element, the random selection process will always return the same control file. As a result of this, each of the four evaluations of the transition expression will invoke OMA using exactly the same input files. Since OMA runs can take a considerable of time, running OMA four times is a complete waste of computing effort. To solve this problem it is necessary to ensure that "OMA" its transition expression is only evaluated if and only if this transition is actually executed for the resulting binding. However, the information about which transitions will be executed for which bindings becomes only available after performing step 3 and can therefore never be used to control the selection process performed in step 1. So the simple execution method provides no means to avoid the unwanted evaluations introduced as a result of its first step.

During the second step of the simple execution method the transitions selected by step 1 are evaluated. During these evaluations *all* the selected transition's net inscriptions are evaluated. For interactive Petri nets this is an highly undesirable situation. As an example again consider the "OMA" transition of Figure 7.12. Now assume that in the current marking the "ctr" place contains a valid control file name, but that the corresponding design file and library name tokens are not available at the "des" and "lib" places. If the Petri net execution method employed specifies that all the net inscription have to be evaluated, then even in this situation the transition expression will be evaluated. As a result of this OMA will be executed even though the required input files are not available. This despite the fact that we already know that the transition can never be executed because the "des" and "lib" places do not contain the tokens specified by "I(des)" and "I(lib)", respectively. This problem can be solved by the introduction of a new transition evaluation method which stops the evaluation the moment it becomes known that the transition can never be executed. This new method should also enable us to control the order in which the net inscriptions are evaluated. Suppose the evaluation of the arc expression of the "ctr" input arc is immediately followed by the evaluation of the transition expression. In this case, the fact that the "des" and "lib" places do not contain the tokens required for the execution of the transition becomes known after the damage has been done. So the new transition evaluation method should enable us to enforce that "OMA" its input arc expressions are evaluated before the transition expression.

During the third step of the simple Petri net execution method, it is determined which of the evaluated transitions will actually be executed. So when this method is used there is no guarantee that an evaluated transition is executed even when the required input tokens are available. In case of the "OMA" transition this means that it is possible that the results of a successful OMA run will be wasted. To avoid problems like this, the execution method used for interactive Petri nets should guarantee that transitions which have been successfully evaluated are always executed.

7.2.5.2 The interactive Petri net execution method

As we have demonstrated in the previous section, the simple coloured Petri net execution method is not suited for the execution of interactive Petri nets. Luckily, the simple execution method is not the only method which can be used for the execution of coloured Petri nets. It is allowed to use an arbitrary other method as long as it leads to Petri net behaviour as specified in Section 7.2.3.2. In this section we are going to define a new Petri net execution method which is customised for the execution of interactive Petri nets.

To solve the blocking problem the new execution method should at least feature concurrent rather than sequential net inscription evaluation. The problem is that real concurrent evaluation is difficult to implement. A much simpler solution is to use a modified form of sequential evaluation, in which the evaluation of a net inscription is suspended the moment it can not proceed due to the evaluation of an interaction construct. In case of a user interaction construct this occurs at the moment it starts waiting for user input. The evaluation of an operating system interaction construct is suspended at the moment it has issued an operating system command. The Petri net execution engine can then (re)start another evaluation. The suspended evaluation can be resumed when either the user has supplied the required input or when the issued operating system command has been completed.

As we have seen in Section 7.2.5.1, one of the causes of the unwanted evaluation problem is that the simple execution method specifies that during the evaluation of a transition *all* its net inscriptions have to be evaluated. This despite the fact that in many cases it is already known after evaluation of some of the net inscriptions that the transition can never be executed. If this is the case, evaluation of the rest of the net inscriptions will be useless. We make use of this fact by adopting a new transition evaluation method referred to as the *lazy evaluation method*. The lazy evaluation method guarantees that net inscriptions are not evaluated when it is known that this is unnecessary. How does the lazy evaluation method determine whether the evaluation of a net inscription is necessary or not? When after the evaluation of an input arc expression the tokens specified by the result are not available at the corresponding input place, then the transition can not be executed for this binding. In this case, evaluation of the remaining net inscriptions becomes unnecessary. A transition can also not executed if its guard evaluates to false. So once it becomes known that the guard is bound to false, the current evaluation of the transition can be stopped.

Even when a net inscription is evaluated, it is not always necessary to do this completely. To illustrate this, consider the expression of the arc connected to the output place "net" shown in Figure 7.12. This expression consists of two parts: the condition "TE = 0 and system("matchStatus 'I(ctr)'") = on" and the expression "stripExt(I(ctr)).oma_net". If the OMA run is unsuccessful or if the control file turns matching off, the condition will fail and there will be no need to evaluate "stripExt(I(ctr)).oma_net". When the lazy execution method is used, then it is guaranteed that such conditional expressions are never evaluated if the corresponding condition fails.

As we have illustrated in Section 7.2.5.1, lazy evaluation is only useful when we are able to control the order in which the net inscriptions are evaluated. A very simple method to control the net inscription evaluation order is by making use of net inscriptions which depend on the results produced by other net inscriptions. Such net inscriptions can only be evaluated after the inscriptions they refer to. As an example again consider the "OMA" transition shown in Figure 7.12. The transition expression of this transition depends on the "I(ctr)" variable, thereby ensuring that it is never evaluated if there is no valid control file name available at the "ctr" input place.

The use of net inscription dependencies can not be used to prevent "OMA" its transition expression from being evaluated before it is known if the required design and library files are available. The reason is that this transition expression does not depend on either "I(des)" or "I(lib)". To solve problems like this we are going to introduce the following default evaluation order. Unless specified otherwise by dependencies, we first evaluate the input arc expressions, followed by the guard, the transition expression and finally the output arc expressions. Using this evaluation order, "OMA" its transition expression will only be evaluated when all the required input file names are available at the input places.

Although the use of lazy evaluation combined with evaluation ordering reduces the amount of unwanted evaluations, it does not completely prevent these evaluations from occurring. To illustrate this again consider the OMA Petri net of Figure 7.12. Suppose now that we have a current marking in which "OMA" its input places contain the tokens required for a single execution of this transition. If in addition the "OMA" transition is selected to be evaluated multiple times before it is actually executed, then each of these evaluations will encounter the required tokens at the input places. This despite the fact that the available tokens only allow us to execute "OMA" once. So except for one, all these evaluations are unwanted.

Even if the "OMA" transition is evaluated only once for the marking described above, there is no guarantee that the transition can actually be executed. The reason is that, in order to solve the blocking problem, we have introduced a more flexible execution method, which allows a transition to be executed the moment it becomes enabled, independent from the status of the evaluation of other transitions. As a result of this, the execution of other transitions can change the contents of the input places of a transition which is being evaluated. So even if the corresponding tokens are available the moment the input arc expressions of the "OMA" transition are evaluated, these can be removed from the input places during the evaluation of the remaining net inscriptions, thereby disabling the execution of this transition. The "OMA" transition is especially susceptible to this, since the evaluation of its transition expression can take a long time.

The two problems described above can be solved by the introduction of what we will call a *reservation mechanism*. Using this mechanism the evaluation of a transition goes as follows. When an input arc expression is evaluated and the tokens specified by the result are available at the corresponding input place, then these tokens will be reserved. In order for a token to be available, it not only has to belong to the contents of the input place, but in addition it may not have been reserved during another transition evaluation. Reserved tokens are either consumed by the execution of the transition reserving them or freed again when the evaluation of the corresponding transition is stopped prematurely. Now again consider the situation featuring a current marking in which "OMA" its input places contain the tokens required for a single execution and where this transition is selected to be evaluated multiple times. During the first evaluation of the "OMA" transition the input tokens required for its execution are reserved. These tokens are not available anymore during the remaining evaluations and these will therefore be stopped before their transition expressions are evaluated. Reservation also ensures that the tokens required for the execution of the "OMA" transition are not removed from its input places by the execution of other transitions during the often long time required for the evaluation of its transition expression. The reason is that in order for a transition to consume a token, it must have reserved it first. Tokens however, can only be reserved if these have not been reserved by another evaluation. Therefore, if the "OMA" transition succeeds in reserving the required input tokens, then these tokens can not be consumed anymore by other transitions.

Based on the solutions to the blocking and unwanted evaluation problem described above, we now define the following interactive coloured Petri net execution method.

1. Select the transition evaluation to be done.
2. Start or continue the selected transition evaluation.
   (a) Select the net inscription evaluation to be done.
   (b) Start or continue the selected net inscription evaluation.
      i. If the evaluation can not proceed due to the execution of an interaction construct, then suspend the current transition evaluation and go to 1.
      ii. If an conditional expression is encountered, then do not evaluate the expression if the condition fails.
   (c) Check whether, given the result of the net inscription evaluation, the transition can still be executed. If not, stop current transition evaluation, release the tokens reserved by this transition evaluation and go to 1.
   (d) If the evaluated net inscription was an input arc expression, then reserve the tokens specified by the evaluation result.
   (e) If all net inscriptions have been evaluated go to 3 else go to 2a.
3. Execute the transition for the binding created by the current evaluation.

Execution of an interaction construct will suspend the current transition evaluation. If this evaluation is selected again later on, then it can either continue by starting the evaluation of a new net inscription or by continuing the suspended net inscription. The latter can only be done if the reason for the suspension no longer exists.

When selecting the next net inscription to be evaluated, the following restrictions have to be considered. When evaluating a transition t, the net inscription n_i has to be evaluated before every other net inscription n_j for which

$$V_{n_i} \in Var(n_j, t) \tag{7.13}$$

holds, where V_{n_i} the variable used to refer to result of the evaluation of n_i. When no such dependency exists between two net inscriptions, then these have to be evaluated according to the principle input arc expressions first, then the guard, transition expression and finally the output arc expressions.

A transition evaluation is stopped when either the tokens specified by an input arc expression are not available at the corresponding input place or the guard evaluates to false. Tokens are said to be available if they belong to the contents of the place and they have not been reserved by other transition evaluations.

The use of the execution method for interactive coloured Petri nets described above eliminates the blocking problem. This execution method achieves this by switching to a new transition evaluation the moment it can not proceed due to the execution of an interaction construct and by allowing transitions to be executed the moment these become enabled.

Our interactive coloured Petri net execution method does not guarantee that unwanted evaluations will not occur. Instead, it provides us with the means to control whether an interaction construct is evaluated or not. Our method guarantees that conditional expressions are not evaluated if the condition fails. In addition, a net inscription is not evaluated if this evaluation is preceded by that of an input arc expression of which the specified tokens are not available or by that of a guard which evaluates to false. Our method guarantees this by stopping the current transition evaluation if such a situation occurs. Finally, the evaluation of interaction constructs in the transition expression and output arc expressions of a transition will never be unwanted if their evaluation is preceded by the evaluation of the corresponding input arc expressions and guard. Our execution method guarantees this because it will always execute a transition if the evaluations of the input arc expressions and the guard were successful. This can be illustrated as follows. After successful evaluation of the input arc expressions we know that the required input tokens were available at the moment these were evaluated. Due to the reservation mechanism, it is guaranteed that this will also be the case at the moment the evaluation of the transition itself is completed. In addition to this, our execution method guarantees that transition evaluations which have been successfully evaluated will always be executed for the resulting binding.

7.3 Design management system representation

In this section we show that a language based on interactive hierarchical coloured Petri nets satisfies our last requirement, i.e., that such a language enables us to create design management system representations according to the DMS model presented in the previous chapters. We do this under the assumption that the DMS is structured as depicted in Figure 3.4. In this case we only have to show that the language satisfies the reduced requirement, i.e., that it enables us to express the design management system representations used to program the DMS core.

The DMS developers program the core by storing the tool, flow and control input descriptions they have created in the design tool store, the design flow store, and the control interpreter, respectively. In the following sections we will show how these tool, flow, and control input descriptions can be expressed in our interactive Petri net based language. Since tools and flows manipulate raw data and communicate using busses, we also demonstrate how raw data and busses can be represented. When a tool or flow description is selected, it is sent to the corresponding execution engine to be executed. Likewise, when a control interpreter receives a control input it will execute the corresponding control input description. Since all these descriptions are expressed using our interactive Petri net based language, the execution engines and the control interpreters can be implemented using the corresponding interactive Petri net simulator.

For design management purposes it is not necessary to create a detailed representation of raw data and tools; usually a much simpler and more abstract description will suffice. We will demonstrate how our interactive Petri net based DMS description language enables us to create these abstract descriptions and to link these to the corresponding raw data or tools. Although raw data and tools do not have to be represented explicitly, the language should enable us to describe how these are controlled by the design management system. For raw data this entails describing how these data are stored and retrieved. For tools this boils down to representing how to execute the tool properly. In contrast to raw data and tools, it should be possible to use the DMS description language to explicitly represent all the details of flows.

The approach we are going to use for the representation of raw data, tools and flows is based on the following considerations. During the design process, raw data are created, modified and deleted using tools. The design flow controls this process. There is a strong analogy between the design process and the way tokens are manipulated by a Petri net. Just like tools transform raw data, Petri net transitions create, modify and delete tokens. Based on this analogy, we propose to represent raw data by tokens and tools by transitions. Just like a flow determines when which tools are applied to what data, a Petri net determines when which transitions are applied to what tokens. Therefore, it is possible to represent flow information in terms of a Petri net.

7.3.1 Raw data representation

If a piece of raw data is represented by a token, referred to as a *raw data token*, then what should the associated value be? In the representation of the OMA tool of Figure 7.12 we represented a piece of raw data by a token with a value equal to the name of the file storing these data. A major disadvantage of this approach is that the design management system will not be able to control access to the corresponding raw data contents. For example, if the contents are stored on a file system, then these data can be modified or deleted by everybody who has the rights to do this. So in this case there is no link between the presence of a raw data token at one of the input places of a transition and the ability of this transition to access the corresponding raw data contents. This can lead to counter intuitive situations. For example, it can occur that a piece of raw data is modified or even deleted while the corresponding token is not processed by a transition at all.

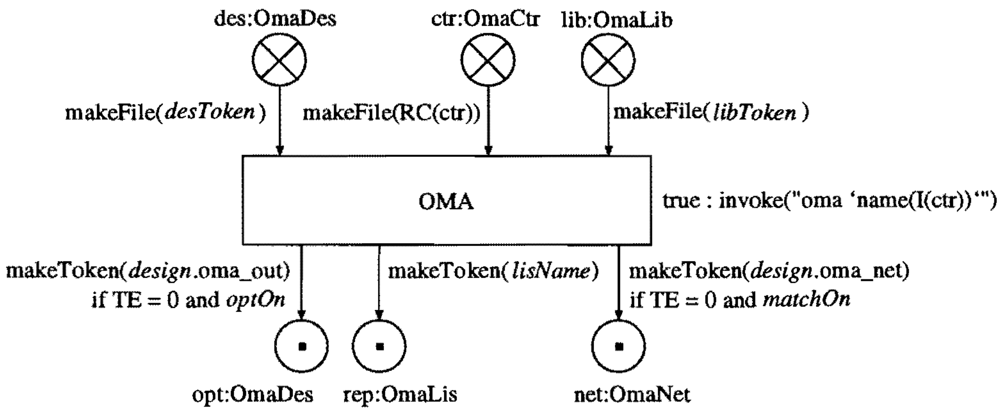
A possible solution for the problem described above is to create a closer link between a raw data token and the corresponding contents, namely by making the value of the token equal to the piece of raw data itself. For this purpose we introduce a new datatype *RawData*. Values of this type, referred to as *raw data values*, store information about both the raw data contents and its properties like its owner, creation time and size.

To access the information stored by a raw data value, we add the following commands to the net inscription language. Information about the properties of a piece of raw data can be retrieved using functions like "name", "owner", "creationTime" and "size", which, when applied to a raw data value, return the name used to identify it, its owner, the time of its creation and its size, respectively. Although it is possible to introduce a function "contents" to return the raw data contents associated with a raw data value, we take an alternative approach. The reason is that the tools operating upon the raw data contents usually access these by reading a file rather than retrieving the contents of a raw data value. To create a link between raw data tokens and the files operated upon by the tools, we introduce the "makeFile(*token*)" operating system interaction construct. Evaluation of the "makeFile(*token*)" construct will use the raw data token *token* to construct a file with the same contents and properties. In addition to reading their input from files, tools usually write their results to output files. To enable our interactive Petri nets to manage these results, we introduce the interaction construct "makeToken(*fileName*)", which when evaluated returns a token with the same contents and properties as the file identified by *fileName*.

Although a raw data token stores the contents of the piece of raw data it represents, it abstracts from information about how it does this. This poses no problems, because these contents are never accessed directly, but by interaction constructs like "makeFile(*token*)" and "makeToken(*fileName*)". If the storage method is changed then only the implementation of these constructs will have to be updated. The only restriction we impose on the contents storage method used for raw data tokens is that it should guarantee that the raw data contents can only be accessed via the corresponding raw data token. For example, if the raw data contents are stored by the creation of a file for each of the raw data tokens, then the access rights of these should be such that these files can only be read, modified and deleted by the Petri simulator creating these.

7.3.2 Tool representation

When raw data are represented by raw data tokens, then a tool can be modelled by a transition manipulating these. Such a transition will consume the raw data tokens representing the tool's input data from its input places. In addition it will add the raw data tokens representing the results produced by the tool to its output places. As an example, consider the interactive CPN transition modelling the operation of the OMA tool shown in Figure 7.13.



design	:=	stripExt(name(I(ctr)))
desName	:=	*design*.system("designExt 'name(I(ctr))'")
desToken	:=	CC(des, name(I(des))) = *desName*)
libName	:=	system("library 'name(I(ctr))'")
libToken	:=	CC(lib, name(I(lib)) = *libName*)
lisName	:=	*design*.oma_lis
optOn	:=	system("optStatus 'name(I(ctr))'") = on
matchOn	:=	system("matchStatus 'name(I(ctr))'") = on

Figure 7.13: Interactive CPN transition representation of OMA.

The "OMA" transition randomly selects a raw data token from the "ctr" input place, which contains raw data tokens representing OMA control files. The file corresponding to the selected token is generated using the "makeFile" command. We now assume that evaluation of the "makeFile" command returns the token it was applied to. As a result of this "makeFile(RC(ctr))" will return the selected control file token. The tokens to be consumed from the "des" and "lib" input places are selected using the Conditional Colour command "CC". When applied to a place p and a condition C, CC(p,C) will return a token randomly selected from p such that I(p) satisfies condition C. In Figure 7.13 this command is used to select a design and a library raw data token from the "des" and "lib" input places, respectively, such that the names of these tokens are as specified by the control file. Like for the control file token, the files represented by the selected design and library raw data tokens are created using the "makeFile" command. Just like the Petri net of Figure 7.12, evaluation of "OMA" its guard will result the invocation of OMA. During the evaluation of the output arc expressions the resulting output files are converted into the corresponding raw data tokens using the "makeToken" command. To keep the representation of the OMA tool simple, we have not modelled the deletion of the files generated during the evaluation of "OMA".

The "OMA" transition of Figure 7.13 will only execute correctly if the input data are of the correct type. For example, if the raw data token selected from the "ctr" input place does not correspond to a valid control file, then application of the "designExt" and "library" programs will produce undefined results. In case of the "OMA" transition this problem is avoided by typing the input data. This is done by labelling the places of this transition with the colour sets "OmaDes", "OmaCtr", "OmaLib", "OmaLis" and "OmaNet". Each time a raw data token is added to a place, the simulator will check whether the corresponding value belongs to the place's colour set. If this is not the case, the simulator will detect an error and will start an error handler to deal with it. Now suppose this handler produces an error message and subsequently stops the simulation. In this case addition of a raw data token to the "ctr" place, which does not correspond to a valid control file, will never result in incorrect execution of "OMA", because simulation will be stopped before this token can even be selected.

The colour sets used to type the places of the "OMA" transition can not be defined by enumeration, because these sets contain an infinite number of elements. For example, the "OmaNet" colour set contains all possible NDL netlists. Therefore, we will not define these sets using enumeration but in terms of a predicate, which determines whether an element belongs to such a colour set or not. As an example again consider the "OmaNet" colour set which is defined as follows

$$\text{OmaNet} = \{\mathit{netlist} \in \mathit{RawData} \mid \mathit{omaNetP}(\mathit{netlist})\}\ , \tag{7.14}$$

where the predicate *omaNetP* is defined by

$$\mathit{omaNetP} = \text{makeFile}(\mathit{netlist});\text{system}(\text{"fileType `name}(\mathit{netlist})\text{'"}) = \text{NDL}\ .$$

When invoked with a filename like "name(*netlist*)" as its argument, the program "fileType" will scan the corresponding file to determine the type. The *omaNetP* predicate is described in terms of a *compound expression*, i.e., an expression consisting of a sequence of smaller subexpressions which are concatenated using the ";" symbol. The value returned by a compound expression is the result of the evaluation of the last subexpression.

The "OMA" transition only controls the execution of a single tool. However, transitions can also be used to model more general tool execution engines like the one shown in Figure 5.2.

For such a transition the tool name is not fixed but received via one of its input places. This name is then used to construct the operating system command used to invoke the tool. This enables us to create DMS descriptions where tools are not explicitly represented by a transition but implicitly by tool information stored in the design tool store.

7.3.3 Representation of busses

Busses are closely related to places. In fact, the operation of a place is equivalent to that of a bus storing an arbitrary number of elements without ordering these. The place's colour set performs a role very similar to that of the bus' contents type. The output and input arcs connected to the place correspond to the sources and drains of the bus, respectively. However, since a place is a special type of bus, it can not be used to represent busses in general. To model busses a concept is required which is more expressive than and at the same time very similar to that of a place. A very promising candidate is the *composite place* concept.

Just like a composite transition, a composite place is described in terms of a Petri net. It is distinguished from a normal place by the fact that it has a fat border. The type of a composite place is given by the name of the corresponding CPN module rather than by a colour set. As an example consider the composite places "opt" and "lib" used in the "OTDM" module depicted in Figure 7.14. The "OTDM" module describes a data driven implementation of the OMA timing driven matching process. It features three composite transitions "optim", "oMatch" and "rMatch", which perform the optimisation, overspecified and realistic timing constraint matching steps, respectively. The corresponding modules "Optimise" and "Match" are implementations of the tool abstractions declared by Equation 5.3 and Equation 5.4, respectively. For simplicity we have not included the descriptions of these modules here, but it is fairly simple to construct these based on the "OMA" transition of Figure 7.13. Furthermore, we assume that "Match" will use a default value of zero when, like in the case of the "oMatch" instantiation, it receives no timing constraint via its "delaySpec" input place.

The specification design and library used during the execution of the "rMatch" transition should be equal to those used by the "oMatch" run preceding it. This can be guaranteed by using a Copying Bus to transport these. The module "CB" of Figure 7.14 implements such a bus. If the output place "out" of a copying bus is empty, then it will randomly select one of the tokens present at its input place "in" and will add two copies of this token to "out".

In this chapter we will restrict ourselves to a simple type of composite place, of which the corresponding module has exactly one input and output place. The semantics of a CPN containing such a composite place is identical to that of a larger CPN, which is obtained by substitution of this place by a copy of the corresponding CPN module. The incoming arcs of the composite place are reattached to the copy of the module's input place. Likewise, the arcs originally originating from the composite place are reconnected to the copy of the

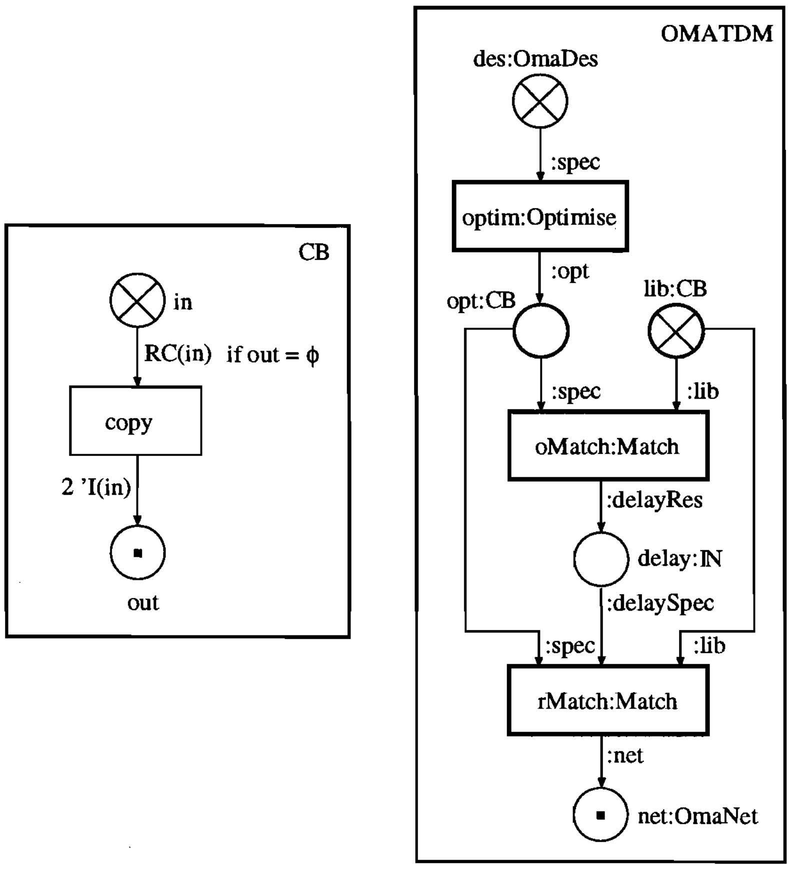


Figure 7.14: The OMA timing driven matching CPN.

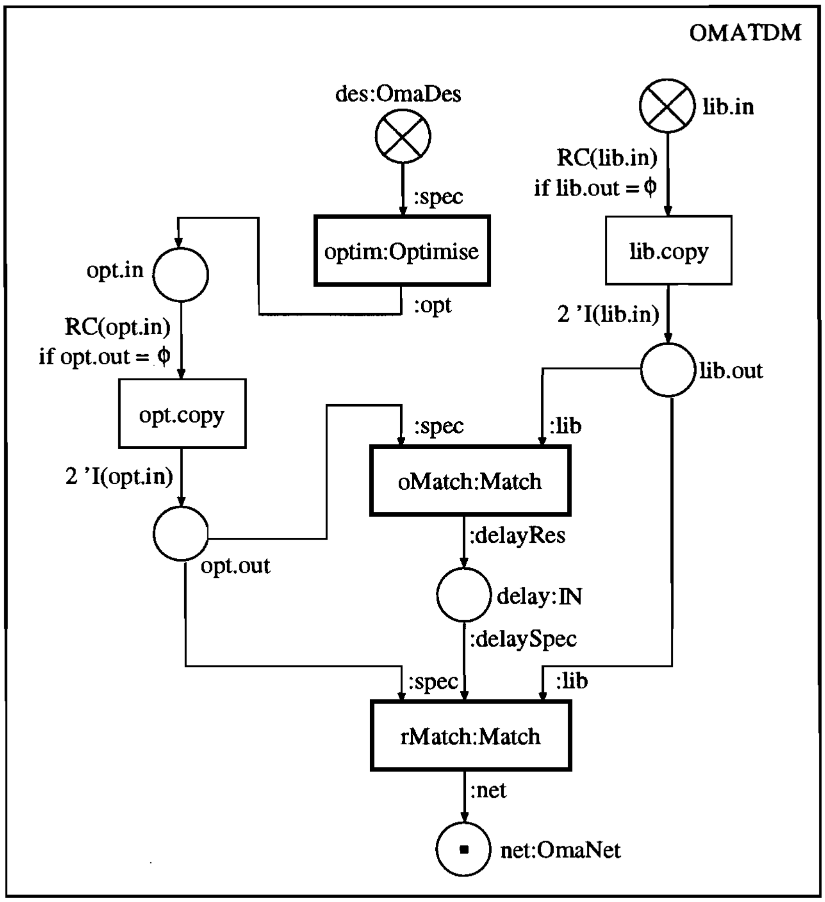


Figure 7.15: The OMATDM module after flattening of "opt" and "lib".

output place. If a composite place is marked as an input or output place, like the input "lib" in Figure 7.14, then the input or output place of the corresponding CPN will also be marked as such after expansion, respectively. When applied to the hierarchical CPN of Figure 7.14, this substitution process will result in the equivalent flattened CPN shown in Figure 7.15. Note that we have maintained place and transition name uniqueness by prefixing the names of the expanded CPN elements with the name of the corresponding composite place.

7.3.4 Representation of flows

Although the operation of a flow can be always be represented by a state machine, there are a number of ways in this machine can be implemented. For example, the flow can be described by a single state machine or by a number of concurrently operating state machines. Likewise, the state machine can be explicitly represented or, like in the case of a data driven flow, very implicitly. A flow representation language should allow flow developers to use each of these different implementation styles. Due to their inherent concurrency, Petri net based language are suitable to represent flows implemented by multiple state machines. As an example of how an implicit flow can be represented by a hierarchical CPN again consider the data driven OMA timing driven matching flow of Figure 7.14. Hierarchical coloured Petri nets can also be used to create explicit flow descriptions. As an example consider the CPN of Figure 7.16, which implements the OMA timing driven matching flow depicted in Figure 6.3. It consists of a place "cs" which contains the current state of the flow, two transitions "nsf" and "of" which represent the next state and the output function, respectively, and three composite places "dtms.stat", "ddms.ctr" and "dtms.ctr" representing the three one place buffers used to monitor and control the DDMS and DTMS. The colour set of the "cs" place "S" is given by Equation 6.1. The "cs" place is initialised with "Opt" which represents that the OMA timing driven matching flow starts in this state.

7.3.5 Representation of control inputs

By programming the control interpreter the DMS developers determine which control inputs it will accept and how these are to be translated into a sequence of design data store inputs. The control input representations they download describe how the control interpreter interprets the abstract control signals it receives from the DDMS's environment by translating these into a sequence of simpler design data store inputs. In addition these representations describe how the control interpreter subsequently uses the resulting design data store output to generate the DDMS output.

As an example of how a control input representation can be expressed using our interactive Petri net based language, consider the CPN of Figure 7.17, which describes how the control interpreter will interpret the "getRootDesign" query. The control interpreter can be instructed to perform this query by sending it the control input pair <getRootDesign, *key*>.

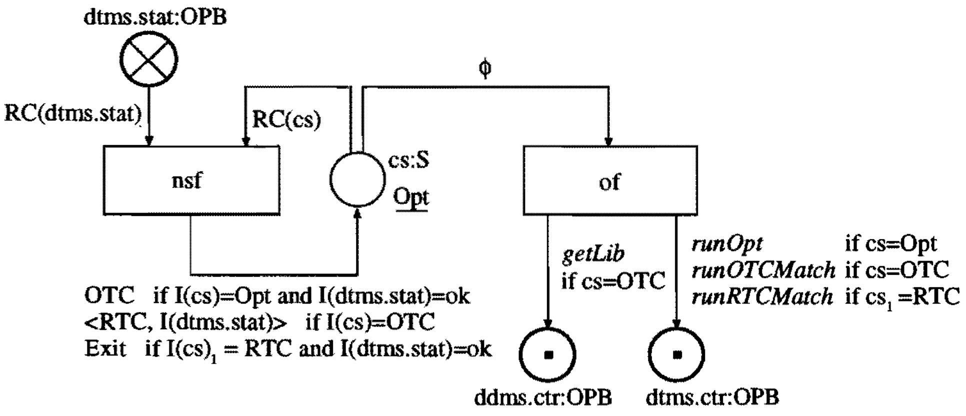


Figure 7.16: Explicit control flow.

In response to this the control interpreter will execute the query described by Equation 4.75, which retrieves all root designs identified by the attribute key *key*. A root design is a design which has been used as a specification to generate an optimised version of it, but which is itself was not created by optimisation of another design.

The CPN of Figure 7.17 features two input places named "Control" and "ddso", which represent the DDMS control input and the design data store output, respectively. In addition it contains the output places "DataOut" and "ddsi", which represent the DDMS data output and the design data store input, respectively.

The CPN of Figure 7.17 features four transitions. The "getKey" transition checks whether the token it has randomly selected from the "Control" input place represents a "getRootDesign" query. If this is the case, it will consume the token and it will send a "Read" command to the design data store instructing it to retrieve all designs characterised by the corresponding attribute key *key*. The "getDesigns" transitions checks whether the status of the resulting design data store output equals "Ok". If this is the case, it will store the retrieved designs in its output place "des". In addition it will instruct the design data store to retrieve all specification designs. The "selectSpecs" transition determines which elements of the design set "I(des)" are specifications by taking the intersection of this set with the specification design set it has received from the design data store. In addition it will issue a "Read" command to the design data store commanding it to retrieve all designs which are obtained by optimisation of another design. The "selectRoots" transition uses

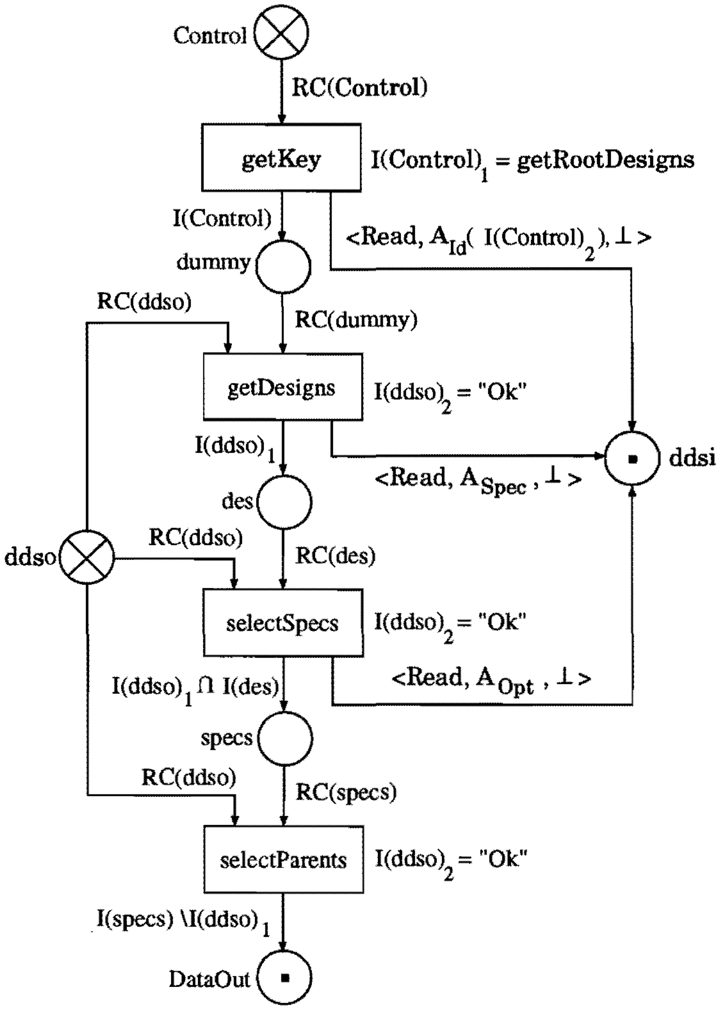


Figure 7.17: CPN representation of the getRootDesign query.

the DDS's response to remove all these designs from the "I(specs)" set and it will send the resulting set of design roots its "DataOut" output place.

7.4 Summary

In this chapter we have demonstrated how to construct a DMS description language, enabling us to create design management system representations according to the DMS model presented in the previous chapters. First we have formulated the requirements this language has to satisfy. Secondly, we have introduced a hierarchical coloured Petri net based language and we have shown that this language already has most of the required characteristics. This language however, lacks support for describing the extensive interaction of a DMS with its users and with the operating system. Therefore, we have extended it with special interaction constructs, resulting in what we refer to as an interactive Petri net based language. When descriptions in this language are executed using the methods normally used by Petri net simulators, then this will result in blocking and unwanted evaluation problems. We have solved this problem by the definition of a new Petri execution method fine-tuned for the execution of interactive Petri nets. A simulator using this method to execute interactive Petri nets will not be bothered by blocking or unwanted evaluation problems. In the last part of this chapter we have demonstrated how our interactive hierarchical coloured Petri net based language can be used to create our executable design management system representations.

Chapter 8

Conclusions and future research

In this thesis we have introduced a method enabling us to *efficiently* develop *complete* CAD frameworks. A complete CAD framework should provide support and a uniform interface to all participants. In this thesis we have shown how this aspect of framework completeness can be implemented. Our approach was based on an analogy between IC designers and design management system developers, which stresses that all these people are designers requiring the same type of support. Based on this analogy we have proposed a new CAD framework structure in which the design management system is the central component, which is used to manage the data, tools, and flows of both the IC designers and the DMS developers. In addition we have introduced a design management system which assists the DMS developers during its own programming process. It achieves this by allowing the DMS developers to program it by downloading DMS representations, rather than by modification of its source code.

The design management system of a complete CAD framework should provide support for all three design management activities. In literature there is little consensus about how such a DMS should be structured. As a result of this most design management system were constructed based on an informal description of (a part of) this system. This is the main reason for the incompleteness and lack of uniformity of these systems. We have avoided these problems by creating a mathematical description of the design management system. This abstract DMS model not only describes the behaviour of the system's three main components, i.e., the design data, the design tool and the design flow management systems, but also shows how these components interact. Using this model as a guideline a complete design management system can be constructed featuring a seamless integration of design data, design tool and design flow management. The feasibility of our model was demonstrated by using it to represent parts of a DMS for the logic synthesis system LOCAM from Philips.

The efficiency of CAD framework development is greatly increased if the corresponding design management system is implemented using a dedicated DMS description language. Since there were no suitable design management system description languages available, we have demonstrated in this thesis how such a language can be defined. Using our DMS model as a guideline, we have first formulated the requirements for such a language. Subsequently we have introduced an extension of the hierarchical coloured Petri net concept referred to as *interactive hierarchical coloured Petri nets*, and have shown how languages based on these interactive Petri nets satisfy our requirements. The suitability of such a language was demonstrated by using it to describe the DMS representations used to program the design management system.

In this thesis we have described the design management system in terms of a design management strategy independent core. This DMS core can be programmed by the design management system developers using a DMS description language such as the interactive hierarchical coloured Petri net based language introduced in this thesis. However, we have not created an implementation of the DMS core, since the required resources exceed those of a one-mans project. Therefore, a topic which can addressed by future research is the implementation of the design management system core.

Although this thesis does not address DMS core implementation, it gives us guidelines about how to do this. In our thesis the DMS core was described in terms of a number of interacting components, namely three stores, three control interpreters, a tool execution engine and a flow execution engine. We have already looked at the hierarchical coloured Petri net design and simulation environment EXPECT [vH94] to see if it can be used as a basis for implementing these components.

The control interpreters and the execution engines perform their job by executing the DMS descriptions provided by the DMS developers. In case the core is programmed using our interactive Petri net based language, these components will behave like a Petri net simulator. So it should be possible to implement these components using such a simulator as a basis. To make EXPECT suitable for our goal, its net inscription language will have to be extended with interaction constructs. In addition EXPECT's execution method will have to be replaced by the interactive Petri net execution method described in this thesis.

The core's three stores use relations to characterise and organise the data they store. For implementing the core's stores EXPECT is again a promising candidate. EXPECT's net inscription language not only features the set and sequence data type required for the construction of relations but also functions to access these. In addition it simplifies the definition of complex relations by allowing its users to define these using diagrams similar to the OTO-D data schemas introduced in Chapter 2. Note however that these features of EXPECT are part of its net inscription language and that it remains to be seen if these can be converted into building blocks for implementing the core's stores.

Appendix A

Raw data distinguishability

We formalise the concept of distinguishability by introduction of a predicate *Distinguishable*(r_1, r_2, C). The r_1 and r_2 arguments of this predicate are either two pieces of raw data or two raw data identifiers. The predicate determines whether r_1 and r_2 can be distinguished in a context C. A context is either a relation or relation element, so either a piece of raw data, an attribute, a raw data identifier, a sequence or a set. For a design data store state *State* *Distinguishable*$(r_1, r_2, \mathit{State})$ determines whether the store is able to distinguish between r_1 and r_2.

Pieces of raw data and raw data identifiers can never be distinguished from themselves, so if $r_1 = r_2 = r$ the following statement holds independent of the context

$$\neg \mathit{Distinguishable}(r, r, C) \ . \tag{A.1}$$

In the remaining part of the definition of *Distinguishable*, we will consider the cases for which $r_1 \neq r_2$ holds.

Pieces of raw data and raw data identifiers are characterised by the way they are related to other data. Without the information about their relation to r_1 and r_2, data on their own are no context which could be used to distinguish between r_1 and r_2. An exception to this rule occurs when r_1 and r_2 are placed in the context of a piece of raw data (raw data identifier) equal to either r_1 or r_2. Due to the implicit relation which exists between such a context and the data to be distinguished, namely that the context is "equal" to one of raw data (identifiers) and "different" from the other, r_1 and r_2 can be distinguished. So if the context C is given by a piece of data d, then the *Distinguishable* function is defined by

$$\mathit{Distinguishable}(r_1, r_2, d) = (d = r_1) \vee (d = r_2) \ . \tag{A.2}$$

Using this definition we can make the following statements about the distinguishability of the raw data identifier "spec" and "opt" (see DDS relation 4.30) in context of the raw data identifiers "spec", "opt" and "rep", the attribute "ALU" and the raw data "*Opt*", respectively.

$$\mathit{Distinguishable}(\text{spec,opt,spec}) \ . \tag{A.3}$$

$$\textit{Distinguishable}(\text{spec,opt,opt}) \;. \tag{A.4}$$

$$\neg\, \textit{Distinguishable}(\text{spec,opt,rep}) \;. \tag{A.5}$$

$$\neg\, \textit{Distinguishable}(\text{spec,opt,ALU}) \;. \tag{A.6}$$

$$\neg\, \textit{Distinguishable}(\text{spec,opt},\textit{Opt}) \;. \tag{A.7}$$

A relation or relation element represented by a sequence T distinguishes r_1 and r_2 if and only if at least one of its elements is able to distinguish between these, so

$$\textit{Distinguishable}(r_1, r_2, T) = \exists i : 1 \le i \le |T| : \textit{Distinguishable}(r_1, r_2, T_i) \;. \tag{A.8}$$

As an example consider the pair <opt,spec> contained in the set representing the "Version-of" relation of DDS relation 4.30. This pair distinguishes between "spec" and "opt", so

$$\textit{Distinguishable}(\text{spec,opt,<opt,spec>}) \;, \tag{A.9}$$

because both its elements do (see Equations A.4 and A.3).

Two pieces of raw data (raw data identifiers) can be distinguished in the context of a relation (element) if these data play a different role in this context. For example <<ALU,OmaDes>,spec> distinguishes "spec" and "opt", because "spec" plays the part of the raw data identifier characterised by key <ALU,OmaDes>, while "opt" does not even occur in this context. Such a relation (element) loses its discriminative power, if it is combined on a basis of equality with another relation (element), which is its equivalent except for an interchange of the roles the raw data (identifiers) play. For example a set with <<ALU,OmaDes>,spec> and <<ALU,OmaDes>,opt> as its elements will not be able to discriminate between "spec" and "opt", this in spite of the fact that both its elements do individually. So the following statement should hold

$$\neg\textit{Distinguishable}(\text{spec,opt},\{\text{<<ALU,OmaDes>,spec>, <<ALU,OmaDes>,opt>}\}) \tag{A.10}$$

For relations represented by a set S the *Distinguishable* function is now defined by

$$\begin{aligned}\textit{Distinguishable}(r_1, r_2, S) = {} & \exists e_1 \in S : \textit{Distinguishable}(r1, r2, e_1) \wedge \\ & \neg\exists e_2 \in S : e_1 \ne e_2 \wedge \textit{RoleSwapEq}(r_1, r_2, e_1, e_2) \;.\end{aligned} \tag{A.11}$$

The function application $\textit{RoleSwapEq}(r_1, r_2, re_1, re_2)$ determines whether the roles played by r_1 with respect to the relation elements re_1 and re_2 are equal to the roles r_2 plays in the relation elements re_2 and re_1, or stated otherwise, if the roles played by the raw data (identifiers) r_1 and r_2 in re_1 have been swapped in re_2. In practice role swap equality requires re_1 and re_2 to be equal except for the occurrences of the raw data (identifiers) in these relation elements. Additionally, it requires that every occurrence of r_1 (r_2) in re_1 is accompanied by an occurrence of r_2 (r_1) at the corresponding location of re_2. Every other piece of raw data (raw data identifier) occurring in re_1 should be matched by a piece of raw data (raw data identifier) at the corresponding position in re_2, which can not be distinguished from it within the context of the DDS state.

Role swap equality is only possible if the corresponding relation elements re_1 and re_2 are of the same type (*RoleSwapEq* returns *false* otherwise). Therefore we will distinguish four cases, corresponding to whether re_1 and re_2 are attributes, raw data (identifiers), sequences or sets, respectively.

In the first case the relation elements re_1 and re_2 are given by the attributes a_1 and a_2, respectively. Since no raw data (identifiers) occur in these attributes, role swap equality requires a_1 and a_2 to be equal. Therefore, in this case role swap equality is defined by

$$\mathit{RoleSwapEq}(r_1, r_2, a_1, a_2) = (a_1 = a_2)\ . \tag{A.12}$$

In the second case re_1 and re_2 are equal to the raw data (identifiers) r_3 and r_4, respectively. Role swap equality implies that if r_3 (r_4) is equal to either r_1 or r_2, then r_4 (r_3) should refer to the remaining piece of data. If r_3 and r_4 are different from r_1 and r_2, role swap equality requires r_3 and r_4 to be equal or indistinguishable within the context of the DDS state *State*. This all is represented by the following definition

$$\mathit{RoleSwapEq}(r_1, r_2, r_3, r_4) = \begin{cases} (r_3 = r_1 \wedge r_4 = r_2) \vee (r_3 = r_2 \wedge r_4 = r_1) & \text{if } r_3 \in \{r_1, r_2\} \vee r_4 \in \{r_1, r_2\} \\ r_3 = r_4 \vee \neg \mathit{Distinguishable}(r_3, r_4, \mathit{State}) & \text{otherwise}\ . \end{cases} \tag{A.13}$$

In case the relation elements are sequences T_1 and T_2, role swap equality is only achieved if all the corresponding sequence elements are again role swap equal. This results in the following definition

$$\begin{aligned} \mathit{RoleSwapEq}(r_1, r_2, T_1, T_2) = {} & |T_1| = |T_2| \wedge \\ & \forall i : 1 \leq i \leq |T_1| : \mathit{RoleSwapEq}(r_1, r_2, T_{1_i}, T_{2_i})\ . \end{aligned} \tag{A.14}$$

Role swap equality of two sets S_1 and S_2 requires every set element to have a corresponding role swap equal element in the other set. This is represented by the following definition

$$\begin{aligned} \mathit{RoleSwapEq}(r_1, r_2, S_1, S_2) = {} & (\forall e_1 \in S_1 : \exists e_2 \in S_2 : \mathit{RoleSwapEq}(r_1, r_2, e_1, e_2)) \wedge \\ & (\forall e_2 \in S_2 : \exists e_1 \in S_1 : \mathit{RoleSwapEq}(r_1, r_2, e_1, e_2))\ . \end{aligned} \tag{A.15}$$

In Chapter 4 we have informally established the (in)distinguishability of the raw data identifiers "spec" and "opt" using the DDS relation rather than the DDS state itself. We will now show why this is allowed. Using the definition of *State* and Equation A.8, $\mathit{Distinguishable}(r_1, r_2, \mathit{State})$ can be rewritten as follows

$$\begin{aligned} & \mathit{Distinguishable}(r_1, r_2, \mathit{State}) \\ & = \mathit{Distinguishable}(r_1, r_2, < \mathit{Raw},\ \mathit{RId},\ \mathit{Attr},\ \mathit{Rel} >) \\ & = \mathit{Distinguishable}(r_1, r_2, \mathit{Raw}) \vee \mathit{Distinguishable}(r_1, r_2, \mathit{RId}) \vee \\ & \quad\ \mathit{Distinguishable}(r_1, r_2, \mathit{Attr}) \vee \mathit{Distinguishable}(r_1, r_2, \mathit{Rel})\ . \end{aligned} \tag{A.16}$$

Using Equation A.2 and Equation A.11 it can easily be shown that *Distinguishable*$(r_1, r_2, Attr)$ will never hold, because none of the elements of the attribute set *Attr* are able to distinguish between r_1 and r_2. In case r_1 and r_2 are both raw data identifiers, the same will hold for *Distinguishable*(r_1, r_2, Raw). For *Distinguishable*(r_1, r_2, RId) there are three possibilities: neither r_1 nor r_2 belong to *RId*, only one of r_1 and r_2 is an element of *RId* and both r_1 and r_2 are contained in *RId*. In the first case neither *RId* nor *Rel* will distinguish between r_1 and r_2, because these raw data identifiers do not occur in these relations. In the second case *RId* will distinguish the raw data identifier occurring in it from the other identifier not belonging to this set. Therefore, *Distinguishable*$(r_1, r_2, State)$ will hold independent from *Rel*. In the last case, in which both identifiers belong to *RId*, *RId* does not distinguish between r_1 and r_2, so *Distinguishable*$(r_1, r_2, State)$ is equal to *Distinguishable*(r_1, r_2, Rel). A similar reasoning holds with respect to the raw data set *Raw* if both r_1 and r_2 are raw data.

In our examples, "spec" and "opt" both occur in the DDS relation, which is only allowed if they also occur in *RId*. So for these examples it makes no difference if we establish the distinguishability of "spec" and "opt" with respect to the DDS relation or the DDS state itself.

In Chapter 4 we have informally argued that "spec" and "opt" can be distinguished within the context of DDS relation 4.30. We will now formally proof that

Distinguishable(spec,opt,*Rel*) (A.17)

holds for this DDS relation, which is given by

Rel =
  {<AttrRIdMap, $\mathcal{P}((Design \times Type) \times RId)$,
    {<<ALU,OmaDes>,spec>, <<ALU,OmaDes>,opt>,
     <<library,OmaLib>,lib>, <<ALU,OmaRep>,rep>}>,
   <Version-of, $\mathcal{P}(OmaDes \times OmaDes)$, {<opt, spec>}>,
   <RIdRawMap, $\mathcal{P}(RId \times Raw)$,
    {<spec,*Spec*>, <opt,*Opt*>, <lib,*Lib*>, <rep,*Rep*>}>} .

The proof that this DDS relation distinguishes between "spec" and "opt" is given by

(1) *Distinguishable*(spec,opt,opt) A.2
(2) *Distinguishable*(spec,opt,<opt, spec>) 1, A.8
(3) *Distinguishable*(spec,opt,{<opt, spec>}) 2, A.11
(4) *Distinguishable*(spec,opt,
  <Version-of, $\mathcal{P}$(OmaDes × OmaDes), *Version-of*>) 3, A.8
(5) ¬ *RoleSwapEq*(spec,opt,
  <Version-of, $\mathcal{P}$(OmaDes × OmaDes), *Version-of*>,
  <AttrRIdMap, $\mathcal{P}$((Design × Type) × RId), *AttrRIdMap*>) A.12, A.14
(6) ¬ *RoleSwapEq*(spec,opt,

        <Version-of, $\mathcal{P}$(OmaDes x OmaDes), *Version-of*>,
        <RIdRawMap, $\mathcal{P}$(RId x Raw), *RIdRawMap*>) A.12, A.14
(7) *Distinguishable*(spec,opt,*Rel*) 4, 5, 6, A.11

Note that we have introduced the shortcuts *Version-of*, *AttrRIdMap*, and *RIdRawMap* for the representations of the "Version-of", "AttrRIdMap", and "RIdRawMap" relations, respectively.

As we have informally demonstrated, the design data store will not be able to distinguish between the input design "spec" and the optimised design "opt" in case of the DDS relation 4.29, which is given by

Rel =
  {<AttrRIdMap, $\mathcal{P}$((*Design* x *Type*) x *RId*),
    {<<ALU,OmaDes>,spec>, <<ALU,OmaDes>,opt>,
     <<library,OmaLib>,lib>, <<ALU,OmaRep>,rep>}>
   <RIdRawMap, $\mathcal{P}$(*RId* x *Raw*),
    {<spec,*Spec*>, <opt,*Opt*>, <lib,*Lib*>, <rep,*Rep*>}>} .

The proof is as follows

(1) *Distinguishable*(spec,opt,<<ALU,OmaDes>,spec>) A.2, A.8
(2) *Distinguishable*(spec,opt,<<ALU,OmaDes>,opt>) A.2, A.8
(3) ¬ *Distinguishable*(spec,opt,<<library,OmaLib>,lib>) A.2, A.8
(4) ¬ *Distinguishable*(spec,opt,<<ALU,OmaRep>,rep>) A.2, A.8
(5) *RoleSwapEq*(spec,opt,<<ALU,OmaDes>,spec>,<<ALU,OmaDes>,opt>)
A.12, A.13, A.14
(6) ¬ *Distinguishable*(spec,opt,AttrRIdMap) A.2
(7) ¬ *Distinguishable*(spec,opt,$\mathcal{P}$((Design x Type) x RId)) Proven later
(8) ¬ *Distinguishable*(spec,opt,*AttrRIdMap*) 1, 2, 3, 4, 5, A.11
(9) ¬ *Distinguishable*(spec,opt,
        <AttrRIdMap, $\mathcal{P}$((Design x Type) x RId), *AttrRIdMap*>) 6, 7, 8, A.8

To complete this proof we still have to verify that statement (7) holds. Because the proof is long but straightforward, we will only present an outline of it here. The type of relation "AttrRIdMap" is the power set of the set (Design x Type) x RId, meaning that it is represented by a set containing all possible subsets of this set. These subsets each group a number of sequences $<< design, type >, rId >$. If one of the subsets contains a sequence $<< design, type >, \text{spec} >$ and not $<< design, type >, \text{opt} >$, then it will distinguish between "spec" and "opt". However, because the type set contains all possible subsets, it will also contain an other subset, which equal to the one presented above, except for the occurrence of $<< design, type >, \text{spec} >$, which has been replace by $<< design, type >, \text{opt} >$. So for every subset distinguishing between "spec" and "opt", there exists another role swap equal subset. The result of this is that the type of relation "AttrRIdMap" is not able to distinguish between "spec" and "opt".

We have now proven that "spec" and "opt" can not be distinguished within the context of the "AttrRIdMap" relation. We continue by showing that this also holds in the context of the "RIdRawMap" relation.

(10)	*Distinguishable*(spec,opt,<spec,*Spec*>)	A.2, A.8
(11)	*Distinguishable*(spec,opt,<opt,*Opt*>)	A.2, A.8
(12)	¬ *Distinguishable*(spec,opt,<lib,*Lib*>)	A.2, A.8
(13)	¬ *Distinguishable*(spec,opt,<rep,*Rep*>)	A.2, A.8
(14)	*RoleSwapEq*(spec,opt,<spec,*Spec*>,<opt,*Opt*>)	
	⇔ ¬ *Distinguishable*(*Spec*,*Opt*,*State*)	A.13, A.14
(15)	¬ *Distinguishable*(spec,opt, *RIdRawMap*)	
	⇔ ¬ *Distinguishable*(*Spec*,*Opt*,*State*)	10, 11, 12, 13, 14, A.11
(16)	¬ *Distinguishable*(spec,opt,<RIdRawMap, $\mathcal{P}$(RId $\times$ Raw), *RIdRawMap*>)	
	⇔ ¬ *Distinguishable*(*Spec*,*Opt*,*Rel*)	A.2, see proof of 7, 16, A.8
(17)	¬ *Distinguishable*(spec,opt,*Rel*)	
	⇔ ¬ *Distinguishable*(*Spec*,*Opt*,*State*)	9, 16, A.11
(18)	¬ *Distinguishable*(spec,opt,*Rel*)	
	⇔ ¬ *Distinguishable*(*Spec*,*Opt*,*Rel*)	Both *Spec* and *Opt* belong to *Raw*

So the raw data identifiers "spec" and "opt" are not distinguishable if and only if the corresponding raw data *Spec* and *Opt* are not distinguishable. The proof of the latter is given by

(19)	¬ *Distinguishable*(*Spec*,*Opt*,<AttrRIdMap,$\mathcal{P}$((Design $\times$ Type) $\times$ RId),*AttrRIdMap*>)	
		Spec and *Opt* do not occur in AttrRIdMap
(20)	*Distinguishable*(*Spec*,*Opt*,<spec,*Spec*>)	A.2, A.8
(21)	*Distinguishable*(*Spec*,*Opt*,<opt,*Opt*>)	A.2, A.8
(22)	¬ *Distinguishable*(*Spec*,*Opt*,<lib,*Lib*>)	A.2, A.8
(23)	¬ *Distinguishable*(*Spec*,*Opt*,<rep,*Rep*>)	A.2, A.8
(24)	*RoleSwapEq*(*Spec*,*Opt*,<spec,*Spec*>,<opt,*Opt*>)	
	⇔ ¬ *Distinguishable*(spec,opt,*State*)	A.13, A.14
(25)	¬ *Distinguishable*(*Spec*,*Opt*, *RIdRawMap*)	
	⇔ ¬ *Distinguishable*(spec,opt,*State*)	20, 21, 22, 23, 24, A.11
(26)	¬ *Distinguishable*(*Spec*,*Opt*,<RIdRawMap, $\mathcal{P}$(RId $\times$ Raw), *RIdRawMap*>)	
	⇔ ¬ *Distinguishable*(spec,opt,*State*)	A.2, see proof 7, 25, A.8
(27)	¬ *Distinguishable*(*Spec*,*Opt*,*Rel*)	
	⇔ ¬ *Distinguishable*(spec,opt,*State*)	19, 26, A.11
(28)	¬ *Distinguishable*(*Spec*,*Opt*,*Rel*)	
	⇔ ¬ *Distinguishable*(spec,opt,*Rel*)	Both "spec" and "opt" belong to *RId*

Equations (18) and (28) allow for two possible solutions: either "spec" can be distinguished from "opt" and *Spec* from *Opt* or "spec" is indistinguishable from "opt" just as *Spec* from *Opt*. So we are not able to prove or falsify $\neg$*Distinguishable*(spec,opt,*Rel*). In cases like this, we will adapt the philosophy that *data are indistinguishable until proven otherwise.* Therefore we will conclude that for DDS relation 4.29 the following statements hold

$\neg$*Distinguishable*(spec,opt,*Rel*) .

$\neg$*Distinguishable*(*Spec*, *Opt*, *Rel*) .

Appendix B

The relation operators

In this Appendix the relation operators introduced in Chapter 4.2 will be formally defined. For the addition to, deletion of and replacement of relation elements we define the *addition operator* $.[.+.]$, the *deletion operator* $.[/.]$ and the *replacement operator* $.[./.]$, respectively. When applied to a relation R, a relation element E and an address A, the addition operator will result in a new relation, denoted by $R[E+A]$, which is obtained from R by the addition of E at the location specified by A. The deletion operator will remove the relation elements selected by an address A from a relation R. The resulting relation is denoted by $R[/A]$. When applied to a relation R, a relation element E and a relation address A, the replacement operator will produce a new relation $R[E/A]$, which is obtained from R by replacing the data stored at the locations specified by A by E.

All these operators only change those elements of a nested relation selected by the corresponding address. All the other elements will not be affected. The element selection procedure is the same for all operators. For a relation R, an element/operator combination EO which equals either $E+$, $/$ or $E/$ and an address given by a sequence of n set or sequence addresses $< A_1, A_2, \ldots, A_n >$, the relation transformation performed by the operator is represented by the following recursive definition. If the relation is represented by a set S, then

$$S[EO \;< A_1, A_2, \ldots, A_n >] =$$

$$\{el | el \in S \wedge \neg A_1(el)\} \cup \{el[EO \;< A_2, \ldots, A_n >] | el \in S \wedge A_1(el)\} \;. \tag{B.1}$$

Likewise, for a relation represented by a sequence T the relation transformation is represented by

$$T[EO \;< A_1, A_2, \ldots, A_n >]_i = \begin{cases} T_i[EO \;< A_2, \ldots, A_n >] & \text{if } A_1(T_i, i) \\ T_i & \text{otherwise} \end{cases} \tag{B.2}$$

for sequence index i $(1 \leq i \leq |T|)$. So the operators will only affect those relation elements selected by set or sequence address A_1. Furthermore, from these elements only those parts selected by the rest of the address $< A_2, \ldots, A_n >$ are changed. The recursion stops when the elements to be changed are selected. For the addition operator the complete address is used to select the set or sequence to which the element is to be added. For the deletion and the replacement operator on the other hand, the address will not select the set or sequence to be changed, but the set or sequence elements to be removed and replaced, respectively.

When the selected element is a set S, then this set is changed as follows. In case of the addition operator, the complete address is used to select S. So the recursion stops when the address sequence is reduced to the empty sequence ϵ. The addition operator will change the selected set by adding E to it. This all is represented by

$$S[E+\epsilon] = S \cup \{E\} \ . \tag{B.3}$$

In case of the deletion and the replacement operator, the set to be changed is selected by the first part $< A_1, A_2, \ldots, A_{n-1} >$ of the address, after which the elements to be deleted or replaced, are selected using the last address A_n of the address sequence. This is reflected by the following definitions

$$S[/A_n] = S \setminus S(A_n), \text{ and} \tag{B.4}$$

$$S[E/A_n] = S[/A_n][E+\epsilon] \ . \tag{B.5}$$

The last definition represents that, in case it is used to change the contents of a set, the replacement operation can be described in terms of the deletion and addition operator.

When the selected relation is represented by a sequence T, the operator definitions become a little bit more complicated. The reason is that sequences not only group their elements, but order them as well. So addition of a sequence element will not only require the element to be added, but also the position at which this new element is to be inserted. Therefore, the element to be added E will be a pair $< pos, E' >$, where pos $(1 \leq pos \leq |T|+1)$ indicates the position at which element E' is to be inserted. Insertion of a sequence element at a certain position will result in an increase by one of both the sequence length and the index of the elements following the newly inserted element. The resulting sequence $T[< pos, E' > +\epsilon]$ is defined as by

$$T[< pos, E' > +\epsilon]_i = \begin{cases} T_i & \text{if } i < pos \\ E' & \text{if } i = pos \\ T_{i-1} & \text{if } i > pos \end{cases} \tag{B.6}$$

for all sequence indices i satisfying $1 \leq i \leq |T|+1$.

Deletion of the elements selected from a sequence T by the sequence address A_n will result in the sequence $T[/A_n]$ defined by

$$T[/A_n]_i = T_j \text{ where } \neg A_n(T_j, j) \text{ and } j = i + |\{T_k | 1 \leq k < j \wedge A_n(T_k, k)\}|, \tag{B.7}$$

representing that the index i $(1 \leq i \leq |T| - |\{T_k | 1 \leq k \leq |T| \wedge A_n(T_k, k)\}|)$ of the remaining elements is obtained from the original index j by decreasing it by the number of deleted elements (read satisfying predicate A_n) originally preceding it.

Replacement by relation element E of the elements selected by the sequence address A_n will result in the sequence $T[E/A_n]$ defined by

$$T[E/A_n]_i = \begin{cases} E & \text{if } A_n(T_i, i) \\ T_i & \text{otherwise} \end{cases} \tag{B.8}$$

for $1 \leq i \leq |T|$, representing that the new sequence was obtained from T by replacing the sequence elements addressed by A_n by the new value E.

Bibliography

[All90] J. Allen. Performance-directed synthesis of VLSI systems. *Proceedings of the IEEE*, 78(2), February 1990.

[AOS94] J. Altmeyer, S. Ohnsorge, and B. Schurmann. Reuse of design objects in CAD frameworks. In *Proceedings ICCAD94*, pages 754–761, San Jose, California, November 1994. IEEE Computer Society Press.

[ASS95] J. Altmeyer, B. Schurmann, and M. Schutze. Generating ECAD framework code from abstract models. In *Proceedings 32nd Design Automation Conference*, pages 88–93, San Francisco, CA, USA, June 1995.

[BC94] R.A. Baldwin and M.J. Chung. Design methodology management using graph grammars. In *Proceedings 31th ACM/IEEE Design Automation Conference*, pages 472–478, San Diego, CA, June 1994.

[BC95] R.A. Baldwin and M.J. Chung. A formal approach to managing design processes. *Computer*, 28(2):54–63, February 1995.

[BD89] M. Bushnell and S.W. Director. Automated design tool execution in the Ulysses design environment. *IEEE Transactions on Computer-Aided Design*, 8(3):279–287, March 1989.

[BD91] J.B. Brockman and S.W. Director. The Hercules CAD task management system. In *Proceedings ICCAD91*, pages 254–257, Santa Clara, California, November 1991. IEEE Computer Society Press.

[BGL92] H. Barringer, G. Gough, and T. Longshaw. A semantics and verification framework for ELLA. Technical report, University of Manchester, 1992.

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1984.

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):1–60, March 1991.

[Bre95] A. Bredenfeld. Cooperative concurrency control for design environments. In *Proceedings European Design Automation Conference*, pages 308–313, Brighton, Great Britain, September 1995. IEEE Computer Society Press.

[BRSVW87] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. MIS: a multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, 6(6):1062–1081, November 1987.

[BtBvW92] P. Bingley, K.O. ten Bosch, and P. v.d. Wolf. Incorporating design flow management in a framework based CAD system. In *Proceedings ICCAD92*, pages 538–545, Santa Clara, California, November 1992. IEEE Computer Society Press.

[BvW90] P. Bingley and P. v.d. Wolf. A design platform for the Nelsis CAD framework. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 146–149, Orlando, Florida, June 1990.

[CFI90] CFI Architecture Technical Subcommittee. *CAD Framework Users, Goals, and Objectives*, version 0.91 edition, Aug 1990.

[CFI91] CFI Design Methodology Management Technical Subcommittee. *Tool Encapsulation Specification Standard*, document dmm-91-g-1, version 0.24 edition, August 1991.

[CFI92] CFI Architecture Tiger Team. *Framework Architecture Reference (Draft Proposal)*, document 91, version 0.88 edition, Januari 1992.

[Dan89] J.D. Daniell. *An object oriented approach to CAD tool control*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213-3890, April 1989. Research Report No. CMUCAD-89-37.

[Das89] S. Dasgupta. *The structure of design processes*, volume 28, pages 1–67. Academic Press, Inc., 1989.

[DD89] J.D. Daniell and S.W. Director. An object oriented approach to CAD tool control within a design framework. In *Proceedings 26th ACM/IEEE Design Automation Conference*, pages 197–202, Las Vegas Convention Center, June 1989.

[Dig91] Digital Equipment Corporation. *PowerFrame Handbook*, 1991.

[EDI93] EDIF Electronic Design Interchange Format version 3 0 0. Reference Manual EIA Standard EIA-618, Electronic Industries Association, 1993.

[Ein89] Eindhoven University of Technology, Faculty of Electrical Engineering, Digital Systems Group (EB). *Structured VLSI Design Course*, 1989.

[FBM94] N. Filer, M. Brown, and Z. Moosa. Integrating CAD tools into a framework environment using a flexible and adaptable procedural interface. In *Proceedings European Design Automation Conference*, pages 200–205, Grenoble, France, September 1994. IEEE Computer Society Press.

[HD96] J.W. Hagerman and S.W. Director. Improved tool and data selection in task management. In *Proceedings 33rd Design Automation Conference*, pages 181–184, Las Vegas, NV, June 1996.

[HNSB90] D.S. Harrison, A.R. Newton, R.L. Spickelmier, and T.J. Barnes. Electronic CAD frameworks. *Proceedings of the IEEE*, 78(2):393–417, February 1990.

[HWS92] U. Hunzelmann, W. Wilkes, and G. Schlageter. Design of a tool interface for integrated CAD-environments. In *Proceedings European Design Automation Conference*, pages 558–563, Hamburg, Germany, September 1992. IEEE Computer Society Press.

[Jan86] A. Di Janni. A Monitor for complex CAD systems. In *Proceedings 23rd Design Automation Conference*, pages 145–151, Las Vegas, June 1986. IEEE Computer Society Press.

[JD92] M.F. Jacome and S.W. Director. Design process management for CAD frameworks. In *Proceedings 29th ACM/IEEE Design Automation Conference*, pages 500–505, Anaheim, California, June 1992. IEEE Computer Society Press.

[JD96] M.G. Jacome and S.W. Director. A formal basis for design process planning and management. *IEEE transactions on computer-aided design of integrated circuits and systems*, 15(10):1197–1211, October 1996.

[Jen92] K. Jensen. *Coloured Petri Nets : Basic concepts, analysis methods and practical use*, volume 1 of *EATCS Monographs on theoretical computer science*. Springer-Verlag, Berlin Heidelberg, 1992.

[KBC+87] R.H. Katz, R. Bhateja, E.E. Chang, D. Gedye, and V. Trijanto. Design version management. *IEEE Design & Test*, pages 12–22, 1987.

[KGMB94] S. Kleinfeldt, M. Guiney, J. Miller, and M. Barnes. Design methodology management. *Proceedings of the IEEE*, 82(2):231–250, February 1994.

[Koo91] C.J. Koomen. *The design of communicating systems : a system engineering approach*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Dordrecht, 1991.

[KT92] E. Kupitz and J. Tacken. DECOR - tightly integrated DEsign Control and ObseRvation. *unknown*, pages 532 – 537, 1992.

[LJ92] D.C. Liebisch and A. Jain. Jessi Common Framework Design Management- The means to configuration and execution of the design process. In *Proceedings European Design Automation Conference*, pages 552–557, Hamburg, Germany, September 1992. IEEE Computer Society Press.

[Meh91] M. Mehendale. An approach to design flow management in CAD frameworks. In *Proceedings of The European Conference on Design Automation (EDAC)*, pages 38–42, Amsterdam, The Netherlands, February 1991. IEEE Computer Society Press.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[Nie86] C. Niessen. *Abstraction requirements in hierarchical design methods*, pages 151–182. Elsevier Science Publishers B.V. (North-Holland), 1986.

[Phi92] Philips Electronic Design & Tools. *OMA v2.5 User Manual*, 1992.

[Rov90a] W.M.H.M. Rovers. Design Flow Description using the EXtended Process Description Language EXPDL. In *Proceedings First WORKSHOP on design, realisation and application of ADVANCED COMPUTERSYSTEMS*, pages 31–58, Faculty of Computer Science, University of Twente, Enschede, The Netherlands, October 1990.

[Rov90b] W.M.H.M. Rovers. A theoretical preliminary study for a Design Assistant. A.I.O. Report AIO-EB-007, Eindhoven University of Technology, Institute for Continuing Education, 1990.

[Rov94] W.M.H.M. Rovers. Description of a Design Management System for the P-ASIC Design Flow using EXPDL. *Formal Methods in System Design*, 4(2):155–166, 1994.

[RRvHK93] W.J.M. Reijntjens, A.K. Riemens, F.G.M. v. Heuven, and J.T.M. Kok. Team work design management concepts and implementation in Cadence Framework II. Nat. Lab. Report 6711, Philips Electronics N.V., 1993.

[RW95] F.J. Rammig and F.J. Wagner, editors. *Glossary*, volume 4, pages 263–280. Chapman & Hall, 1995.

[SBD93] P.R. Sutton, J.B. Brockman, and S.W. Director. Design management using dynamically defined flows. In *Proceedings 30th ACM/IEEE Design Automation Conference*, pages 648–653, Dallas, Texas, June 1993.

[Sch94] O. Schettler. Design tool encapsulation - All problems solved? In *Proceedings European Design Automation Conference*, pages 206–211, Grenoble, France, September 1994. IEEE Computer Society Press.

[Sch95] O. Schettler. *Encapsulating Tools into an EDA Framework.* PhD thesis, Delft University of Technology, September 1995.

[SD96] P.R. Sutton and S.W. Director. A description language for design process management. In *Proceedings 33rd Design Automation Conference*, pages 175–180, Las Vegas, NV, June 1996.

[Sie91] E. Siepmann. *Entwurfstheorie und Entwurfsdatenmodellierung fur CAD-Frameworks.* PhD thesis, University of Kaiserslautern, September 1991. D 386.

[Sim93] M.N. Sim. *Tool and Database Interfacing based on Virtual Objects.* PhD thesis, Delft University of Technology, January 1993.

[Sto91] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Eindhoven University of Technology, July 1991.

[tB92] J. ter Bekke. *Semantic Data Modeling.* Prentice Hall. Addison-Wesly Publishing Company, UK, 1992.

[tB95] O. ten Bosch. *Design Flow Management in CAD Frameworks.* PhD thesis, Delft University of Technology, September 1995.

[tBBvW91] K.O. ten Bosch, P. Bingley, and P. v.d. Wolf. Design flow management in the Nelsis CAD framework. In *Proceedings 28th ACM/IEEE Design Automation Conference*, pages 711–716, San Francisco, California, June 1991.

[TY86] T. Tomiyama and H. Yoshikawa. Extended general design theory. Report CS-R8604, Centre for Mathematics and Computer Science, 1986.

[vH94] K.M. van Hee. *Information Systems Engineering: A Formal Approach.* Cambridge University Press, 1994.

[VHD93] VHDL language reference manual. IEEE Standard 1076, IEEE, 1993.

[vHL96] P. v.d. Hamer and K. Lepoeter. Managing design data: The 5 dimensions of CAD frameworks, configuration management and product data management. *Proceedings of the IEEE*, 84(1), January 1996.

[vHT90] P. v.d. Hamer and M.A. Treffers. A data flow based architecture for CAD frameworks. In *Proceedings ICCAD-90*, pages 482–485, Santa Clara, California, November 1990. IEEE Computer Society Press.

[vHtBvLvW94] A. v.d. Hoeven, O. ten Bosch, R. van Leuken, and P. v.d. Wolf. A flexible access control mechanism for CAD frameworks. In *Proceedings European Design Automation Conference*, pages 188–193, Grenoble, France, September 1994. IEEE Computer Society Press.

[vW93] P. v.d. Wolf. *Architecture of an Open and Efficient CAD Framework*. PhD thesis, Delft University of Technology, June 1993.

[vW94] P. v.d. Wolf. *CAD Frameworks: Principles and Architecture*. Kluwer Academic Publishers, 1994.

[vWBD90] P. v.d. Wolf, P. Bingley, and P. Dewilde. On the architecture of a CAD framework: The Nelsis approach. In *Proceedings of the European Design Automation Conference*, pages 29–33, Glasgow, Scotland, March 1990.

[vWSBD90] P. v.d. Wolf, G.W. Sloof, P. Bingley, and P. Dewilde. Meta data management in the Nelsis CAD framework. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 142–145, Orlando, Florida, June 1990.

[Yos81] H. Yoshikawa. General design theory and a CAD system. *Man-Machine Communication in CAD/CAM*, 1981.

[ZG96] M. Zanella and P. Gubian. A conceptual model for design management. *Computer-Aided Design*, 28(1):33–49, 1996.

Curriculum Vitae

Willem Rovers was born on 3 July 1963, in Gemert, the Netherlands.

He attended the Macropedius College in Gemert from 1975 to 1981. In September 1981 he started to study Physics at the Eindhoven University of Technology. He performed his graduation research at the Theoretical Physics group, where he worked on a topic involving the quantum mechanical description of collisions of hydrogen atoms at low temperatures. In November 1987 he received his master's degree.

In February 1988 he started to work under the supervision of prof. dr. ir. C.J. Koomen as a research assistant at the Faculty of Electrical Engineering. His task was to create an expert system to support the design of digital systems. In April 1990 he received the Master of Technological Design degree in Information and Communication Technology from the Stan Ackermans Institute at the Eindhoven University of Technology. From 1990 to 1992 he was sponsored by Philips to perform Ph.D. research in the area of design management and CAD frameworks.

Since September 1992 he has been working as a research scientist at Philips Research in Eindhoven, where he has been doing research on the topic of formal functional verification of IC design. This has resulted in the functional equivalence checker YATC which is now extensively used by the design community of Philips.