53
Management Application Creation with DML

Barbara Fink, Helmut Dercks, Peter Besting
Philips GmbH - Research Labs
P.O.Box 1980, D-52021 Aachen, Germany
Tel.:+49.241.6003-509 Fax:+49.241.6003-519
E-mail:fink@pfa.philips.de

Abstract

This paper presents the current state of the DOMAINS Management Language (DML) which was in its first version developed in the ESPRIT project DOMAINS and enhanced thereafter. DML is an extension of the ISO standard GDMO offering a formal and executable behaviour. The language features, the corresponding compiler and the embedding management architecture are explained. In addition, experiences gained with employing DML for non-trivial applications is reported on. Although DML has not yet reached full maturity, it is a very useful tool that successfully assists application developers. The approach of combining a specification language with an implementation language proved to be very helpful: It allowed to use already standardised GDMO specifications and to convert them into executable programs with relatively little programming effort.

Keywords

Network management, management language, managed object, management application creation, DOMAINS, DML, GDMO, GDMO compiler.

1. INTRODUCTION

The market for network systems is rapidly growing, and the increasing complexity of network systems calls for a well structured management system consisting of a generic management platform and individual applications. In order to facilitate the efficient development of applications independent of the underlying platform, a management language is needed that

- provides appropriate high-level expression means to the management application programmer for efficient and reliable application development,
- hides application irrelevant concepts and the implementation of the underlying software and hardware components, and that
- can be translated automatically into an executable program.

A first step meeting the first two requirements was made with the ISO/IEC standard "Guidelines for the Definition of Managed Objects - GDMO"[1]. However, GDMO focuses on specification in contrast to implementation. The current standard is restricted to module interface and structuring descriptions, whilst the managed object's semantic, i.e. the behaviour description, is postponed. Current GDMO applications typically wrap the behaviour as plain English text in comments. Recent standardization efforts discuss to use Formal Description Techniques - e.g. SDL [2], Z, VDM, or LOTOS [3] - for the behaviour. The GDMO extension

A. S. Sethi et al. (eds.), *Integrated Network Management IV*

LOBSTER [4] attempts, as well, to integrate formal behaviour parts into the GDMO. It is based on extended CRS (Communicating Rule Systems). Here, the behaviour of a MO (Managed Object) is defined as the sequence of all observable interactions with its environment. All these approaches focus primarily on rigorous specification without concern of the final implementation. In contrast the tool DAMOCLES [5] is more technique oriented. It contains a MO Browser which gives a structured overview of all existing MO Classes and a GDMO Template Editor which guides the programmer in writing syntactically correct and semantically consistent GDMO specifications. However, none of these approaches achieves automatically generated executable programs.

It is commonly agreed that there is a strong and increasing demand for the formalization of GDMO behaviour. In addition, the authors believe that the method to be used should allow automatic, unambiguous translation into executable code which can run on different target platforms. This latter requirement is considered extremely important as there are already various standardized specifications in GDMO (as e.g. the Generic Network Information Model [6] or the SDH NE Information Model [7]), the implementations of which should result in identical effects when being used and controlled by different management systems.

Motivated by the reasons stated above and last but not least by the need for efficient management application creation, the high level management language DML (DOMAINS Management Language) was developed. It was in its first version developed in the scope of the ESPRIT Project 5165 DOMAINS (cp. [8], [9] and [10]), enhanced continuously thereafter and extensively used for various applications.

The following chapter gives an overview of the management architecture containing DML. We then introduce the language and its compiler followed by experiences gained when using the language for non-trivial applications. Finally we discuss future enhancements and still open issues.

2. THE EMBEDDING MANAGEMENT ARCHITECTURE

2.1 The Management Model

The embedding management architecture goes back to the DOMAINS project mentioned above. One of its basic principles enhances the OSI Manager-Agent model by the concept of *domains*: Domains are used to recursively decompose the overall management task into subtasks. A domain comprises a manager and the set of resources to be managed. Depending on the complexity of the management task, a managed resource can be a simple real resource or again an entire lower level domain. This way domains are used to build up a management hierarchy. The manager at the top of the system plays the manager role in accordance to the OSI manager. Managers at the bottom controlling real resources can be seen as agents in the sense of OSI. The managers on the intermediate levels control managers on a lower level while at the same time being managed by those on a higher level. Taking the recursiveness into account, both managed and managing components must be treated uniformly: DOMAINS introduced for their representation the concept of the *Kernel*.

Due to a possible overlap of domains, resources may be controlled by several managers. This leads to an m:n relationship between managers and resources where, in general, different managers have different views of one and the same resource. This idea is supported by the *Shield* concept. Whereas the Kernel represents the complete behaviour of a manager or resource, a Shield represents an interface only and that precisely tailored to the needs of the superior manager.

2.2 The Management Platform

This section describes the overall management platform, which DML is a part of. As depicted in Figure 1 the application independent stack consists of the hardware, an operating system, a distributed processing system, the DOMAINS machine and finally the DML language with its compiler. In our implementation, ANSAware of APM[1], itself residing on UNIX, is used as basis for the DOMAINS machine. Whereas ANSA provides distribution transparency and basic communication facilities the DOMAINS machine adds specific functionality such as services for event handling or notification registration. In addition, the DOMAINS machine supports dynamic object class and instance creation.

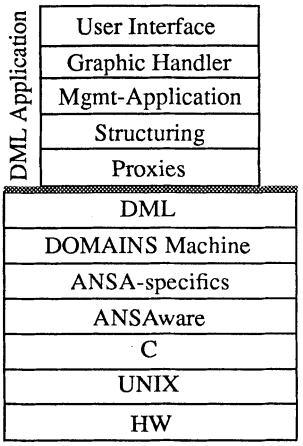

Figure 1: Management Platform

The DML compiler translates DML programs in the ANSA programming language IDL/DPL enriched with function calls for specific DOMAINS services.

A typical DML application is structured itself in several layers: The proxies are a collection of predefined specifications normally agreed upon by standardization committees. To guide the developers in designing management systems, structuring guidelines [11] have been developed. These define different views of the Management Information using the management model outlined in section 2.1. The management application - a complete specification of a management problem in DML - has to rely on these design guidelines and structuring principle. Finally, the graphical user interface can be seen as a specialized manager, who has communication paths to (possibly) all other managers in the system.

3. LANGUAGE FEATURES

3.1 Principles

DML's primary goal is to provide upward compatibility to the ISO standard GDMO to the greatest possible degree. Minor deviations were accepted in order to achieve a first running version within a given time schedule.

We start with a brief review of the basic GDMO features. Managed objects are specified by

- *Attributes* determining the object's state,
- *Actions* that can be coerced by managers through invocations, and
- *Notifications* that are issued by the managed objects to indicate, for example, attribute value changes.

From these features *Packages* can be built which in turn can be used as the building blocks of *Managed Object Classes*. A set of *templates* give proformas for specifying these features according to their external view.

1. ANSAware is a trademark of APM Architecture Projects Management Limited, Cambridge

In GDMO the formal specification is restricted to syntax aspects. DML realizes extensions with respect to the application scope and the semantics.

Managed and managing objects

The standard considers only management targets, i.e. managed objects, whereas the management activities exercised by managers are not treated. In contrast, the recursive DOMAINS management model - according to which a managed object may itself exercise management control on lower level managed objects - requires a common model for both managed and managing objects. Thus DML supports the description not only of managed resources but of managers as well. This puts extended requirements on the expression power of the behaviour clauses.

Different kinds of object classes

DML supports the DOMAINS Management Architecture by introducing different kinds of object classes, i.e. *Kernel* -, *Shield* - and *Support Object Classes* (cp. Section 2.1).

Operational and declarative behaviour language

As mentioned earlier a formal GDMO behaviour description is currently still missing. Therefore the new language was enhanced by an operational behaviour language yielding a general-purpose object-oriented programming language. For special purposes - cp. notifications in Section 3.5 - also declarative behaviour description is supported.

Integrating ASN.1

GDMO employs the ISO standard ASN.1 [12] for the description of data types, however, as semantics is not treated at all, the notation for accessing variables and/or their substructures is omitted. DML, now, has integrated a subset of ASN.1 into the behaviour language supporting convenient and type-save access to ASN.1 data.

3.2 Data Types

In accordance to GDMO, data types are distinguished from object classes. They conform to the ASN.1 standard. Conceptually DML comprises the full set of ASN.1, though the current language version is restricted to a subset only, comprising the entire set of simple types - e.g. BOOLEAN, INTEGER -, structures (SEQUENCE) and lists (SEQUENCE OF). With the intention to increase program reliability, untyped pointers or ANY are not supported in DML.

3.3 The DML Object Classes

In order to sufficiently support the management model outlined in section 2.1, DML distinguishes between three kinds of object classes:

- *Kernel Object Classes*
- *Shield Object Classes*
- *Support Object Classes.*

Kernel objects are used to represent managers and managed resources. Shield objects may represent Shields only that are inserted between a manager and a managed resource. From the language's point of view the Shield object has restricted functionality as compared to the Kernel object. Most of the Shield object's functionality is transparent to the application programmer. Its essential function is forwarding invocations and notifications. In the case of external

resources residing in foreign systems, protocol transformations may be involved, hidden to the application programmer. However, in the current implementation protocol transformations are not realized. Support objects are foreseen for auxiliary tasks, such as mathematical functions, data base handling.

3.4 Object structuring

GDMO and thus DML offers various concepts and techniques for structuring objects.

Templates

The technique of templates serving as building blocks and supporting code re-usability is unrestrictedly adopted from GDMO. In addition we also allow inline-coding of templates. This method supports the traditional inline block-structured programming style. It is preferably used if otherwise control over a great number of small separate templates would be lost.

Inheritance

DML supports multiple inheritance as does GDMO. However, our current implementation does not inhibit repeated inheritance, i.e. there is no check if one and the same template is inherited several times.

Object References

Object structuring may also be achieved according to client/server modelling. In DML, objects can be accessed location transparently by their user-given name or by a typed variable that contains an object reference.

Polymorphism

The strong typing concept with static type checking supports dynamic binding that copes with polymorphism. The DML polymorphism concept is based on inheritance analogous to Eiffel [13], i.e. any inheriting class can be taken as its base class.

Action Templates

There are three ways to define actions: by direct, *deferred* or *external* specification. Deferred actions are adopted from Eiffel. The behaviour specification of these actions has to be specified in the inheriting classes. External actions are provided to link foreign programming languages to DML. Currently C is being supported.

3.5 Inter Object Communication

Objects interact with each other by means of *Invocations* and *Notifications*. The basic difference between these two types is the addressing concept: Invocations are explicitly addressed to their final destination, where they implicitly activate the corresponding *action*. In contrast, spontaneously emitted notifications do not know their final destination. One or several interested objects may register for certain notifications. Thus the destination object must take initiative for receiving a notification.

DML has introduced the concept of *Notification Handlers* specifying the reactions upon received notifications.

Actions are executed due to invocations and notification handlers due to notifications.

Invocations are sent from objects in the *manager* role to objects in the *resource* role for the purpose of *controlling* resources. The notification flow is in opposite direction, it is used for *monitoring* resources.

Invocation Types

DML distinguishes between

- synchronous, blocking invocations, called *CALL,*
- synchronous, non-blocking invocations, called *FORK*, and
- asynchronous, non-blocking invocations, called *CAST.*

All three types can pass arguments to their destination. The first and second one support reply arguments as well. In the case of a CALL the invoking program thread is suspended until the reply is received, whilst after a FORK and CAST the program thread is immediately continued, resulting in concurrently running actions. Any time after having issued a FORK invocation, the invoker can request the reply.

Notification Types

Notifications can pass arguments to the receiver(s). Unlike GDMO, DML does not support confirmed notifications. Reply parameters cannot be returned. In this case DML's restriction with respect to GDMO was deliberately undertaken. Notification confirmation is not considered necessary in the employed management model.

Notification emission specification can be

- imperative by the *NOTIFY* command or
- declarative by a logical expression over attributes.

As soon as the logical expression becomes true, the corresponding notification is emitted. This way attribute value change notifications can be naturally specified. The current implementation does not support declarative notification specifications.

Notification Registration

As stated above, objects playing a manager role must register for notifications in order to receive them. Selection criteria are the notification type, the emitting object class or object instance. In this way a manager may register for a certain notification type regardless of its source, or for a certain notification type sent by all instances of a certain class, or for a certain notification type sent by a certain object instance. The registration is dynamic, it can be cancelled again.

The registration command denotes also the notification handler, i.e. the program piece that is to be executed upon reception of the notification.

3.6 Attributes

Attributes are part of the external interface. They are accessible from other objects according to specified operations as e.g. GET or REPLACE. This aspect corresponds to the GDMO standard.

Additionally, attributes must be related to the object's own behaviour. From the object-internal view attributes are common data with full visibility according to their type. Whilst object-external access is restricted to the attribute as a whole, the object itself may access also individual data components and perform operations on them - e.g. multiplications - as defined for the specific type.

For denoting individual data components the familiar dot-notation and/or bracketed indices are applied.

3.7 Behaviour Description

DML comprises a general-purpose behaviour language. With the requirement "easy to learn and easy to use" it contains only very few and safe constructs. Eiffel was taken as a model for the notation of expressions, assignment-, conditional- and loop-statements. Transactions and special statements for object interaction as mentioned in section 3.5 are added.

Object-common data were already mentioned in the previous section. We introduced also local data for individual behaviour templates to be used as temporary local working variables.

3.8 Assertions

DML supports runtime semantics checks. There are built-in default checks - e.g. on list bounds - as well as user-defined assertions. For the latter the Eiffel concept is adopted: Action-behaviours can be enhanced by asserted pre- and post-conditions. User-defined *exception handlers* are executed if the assertions are violated.

3.9 Example

This section presents extracts from a DML program listing. The Fabric object selected represents the switching unit in a transmission network node. Its basic task is to control the set-up and release of cross-connections between pairs of termination points. Most of the program is self-explaining, some extra comments (beginning with a double hyphen) were added for convenience.

```
--              *** Fabric.dml ***
-- These are instructions for the pre-processor to include certain files.
USE "DML_Standard"  -- This file contains DML standard definitions etc.
USE "TypeDefs"      -- TypeDefs contains general ASN.1 type declarations.
USE "ProxyMO"       -- This one is used for inheritance.
USE "Adapter"       -- The Adapter object is the link to the managed network.
...

--              *** Fabric KERNEL Template ***
Fabric KERNEL OBJECT CLASS
    DERIVED FROM ProxyMO;
    MANAGING PART CHARACTERIZED BY fabricManagingPackage;
;
fabricManagingPackage PACKAGE
    ATTRIBUTES
        tpPool              GET,
        crossConnections    GET,
        adapterName         GET,
        ...
    ;
    ACTIONS
        connect,
        disconnect,
        ...
    ;
;
```

```
--          *** ATTRIBUTE Templates ***
tpPool              ATTRIBUTE WITH ATTRIBUTE SYNTAX MOIds;;
crossConnections    ATTRIBUTE WITH ATTRIBUTE SYNTAX XConnections;;
adapterName         ATTRIBUTE WITH ATTRIBUTE SYNTAX OCTET STRING;;
...;;

--          *** ACTION Template ***
connect ACTION
   BEHAVIOUR connectBeh BEHAVIOUR DEFINED AS
   @          -- Identifies beginning of our formalised behaviour extending GDMO.
      VARIABLES
         iloop          INTEGER,
         loopFlag       BOOLEAN,
         connectRequest Request,        -- Request is declared in TypeDefs.
         connectReply   Reply,          -- Reply is declared in TypeDefs.
         stdReply       StandardReply,  -- StandardReply is declared in TypeDefs.
         adapterRef     Adapter,        -- Object references are declared in this way.
         ...
      ;
      DO
         ...
--             *** fill message structure and send request to adapter ***
         connectRequest.modsimMsg[0] := "connect";    -- Assigning a value to a data structure.
         connectRequest.modsimMsg[1] := xConnection.from.instance;
         connectRequest.modsimMsg[2] := xConnection.to.instance;
         adapterRef := adapterName;                   -- Assigning a value to an object reference.
         CALL adapterRef.sendRequest(connectRequest ->connectReply);      -- Action invocation.
         ...
--             *** remove the "to" tp from the tpPool ***
         loopFlag := TRUE
         FROM iloop:=0;
         UNTIL (iloop >= LENGTH(tpPool)) OR (loopFlag = FALSE)
         LOOP
            IF tpPool[iloop].instance = xConnection.to.instance
            THEN
               REMOVE(tpPool[iloop]);               -- Predefined access method REMOVE.
               loopFlag := FALSE;
            ENDIF;
            iloop := iloop + 1;
         ENDLOOP;
         ...
         RETURN stdReply;
      END      -- End of DO range.
   @;          -- Identifies end of our formalised behaviour extending GDMO.
   ;           -- End of BEHAVIOUR
   WITH INFORMATION SYNTAX xConnection : XConnection;-- ACTION input parameter
   WITH REPLY SYNTAX StandardReply;                  -- ACTION reply parameter
;              -- End of ACTION
...
```

The implementation shown here follows the specification of the fabric object according to the ITU standard M.3100 [6].

4. LANGUAGE ARCHITECTURE AND COMPILER

The DML language incorporates and integrates the GDMO specification language, the ASN.1 notation and declarative and procedural statements to express the behaviour. It allows the complete specification of management applications, that - once compiled - are executable.

The template-oriented language supports piece-wise compilation. The compilation unit is a DML file and the programmer is free to collect several templates into one DML file.

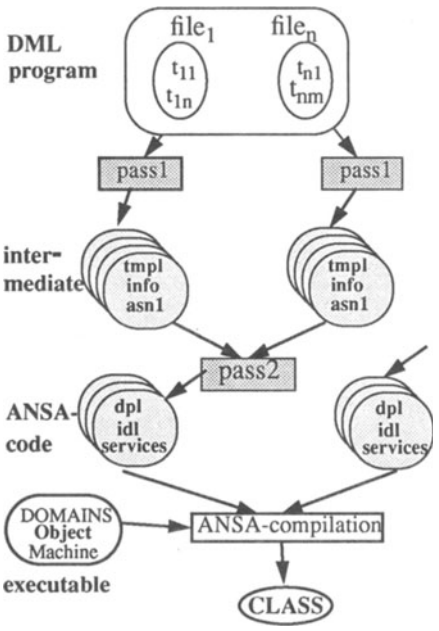

Figure 2: Compiler Structure

The compiler analyses the template descriptions and stores them in an internal repository. Data structures, described in ASN.1 notation, are mapped to C structures. Access, manipulation and assign functions are automatically generated for them.

The templates are bound together to object classes, which can be instantiated during run-time. The necessary anchors to compose the templates are stored in so called *info files* (one per object class).

The compiler is composed of two passes (cp. Figure 2). The first one is responsible for syntax checks. It builds the specific template files, ASN.1 mappings and the info files. The second pass is dedicated to semantic checks of templates and packages and their inter-relation.

The backend part generates code in the ANSA interface- and programming language IDL and DPL. It also produces support files for memory allocation/deallocation and ASN.1 data handling. The ANSA compiler takes care of processing DPL and IDL files and linking the output with earlier generated service routines to a complete class description that can be started and instantiated by the DOMAINS machine (cf. section 2.2).

5. EXPERIENCES WITH DML

This section presents first-hand experiences with DML that were gained during the development and test of several management applications in different scenarios. DML significantly facilitates management application creation during the specification and implementation phase. The following subsections provide detailed evidence for this statement, but they also point out the main handicaps that have to be overcome in future versions of the language.

5.1 Application Scenarios

The three major DML application scenarios we refer to are related to the management of the freephone services in an intelligent network, to a fault, configuration, and service management system for an SDH network, and to the management of an ATM switching system. The SDH application resulted in a system that was presented to the public on the last CeBIT fair in Hanover in March '94. It is structured in 50 object-classes described in about 20.000 lines of DML code.

5.2 User Friendliness

DML is user friendly from various points of view:

- Short learning period of only few and simple but powerful basic constructs for data representation and control statements. Complex data structures can be accessed via a familiar point and index notation.
- Uniform programming style enforced through predefined template structures.
- Self-documentation and good readability of the program code.

5.3 Safer Code Production

The features that guarantee a safer code production than is achieved with other programming languages like C or C++ can be grouped along four main aspects:

- Raising the application programming abstraction level and freeing the application programmer from routine tasks like memory allocation and de-allocation.
- Reduction in number of code lines by an order of magnitude due to the high abstraction level.
- Restriction to safe language constructs and strong typing.
- Runtime semantics checks e.g. to prevent array overflow or use of null references.

5.4 Integration of Standard Specifications

Industrial organizations (like the ATM Forum) and standardization bodies (like ITU and ISO) put much effort into the design of open interfaces and standardized information models. These models are specified along the guidelines of GDMO. Due to DML's GDMO compatibility, these specifications can serve directly as a first code version. What is left over is the behaviour which is just given as comments in plain English and which has to be replaced by corresponding DML code. This extended GDMO code is then fed into the compiler to produce the executables. Compared to other approaches where the GDMO specifications are first translated into e.g. C or C++ code which then has to be extended with C or C++ behaviour parts, our correspondence of specification language and implementation language guarantees a much smoother program development process.

5.5 DML shortcomings

Using DML in practical applications also revealed some of its limitations and disadvantages. First to mention is that not all of the originally designed language features are supported by the current compiler.

A nuisance is the excessive use of semi-colons which terminate declarations, statements, packages, etc. which, however, is prescribed by GDMO.

A more serious shortcoming is the only very basic input and output functionality that is currently provided. And finally, testing and debugging is not yet sufficiently supported.

6. FUTURE ENHANCEMENTS

Desired enhancements can be grouped according to activities concerning the language definition and compiler and to the tools supporting the application programmer.

Language definition

New and/or enhanced concepts to be developed comprise:

- object persistency,
- combination of enhanced declarative and imperative description methods,
- intelligent notification filters,
- notion of time.

Tools

A window-oriented template editor should guide and assist the programmer in writing syntactically correct applications. A still open issue is an adequate debugging tool suited for a distributed environment.

7. CONCLUSION

DML is a high level management language that extends GDMO with a formal and executable behaviour. Experiences gained with several applications showed that DML significantly simplifies management application creation during the specification and implementation phases.

The main conclusions from these experiences are:

- DML has evolved into a useful tool
- DML is extremely user-friendly
- DML supports safer code production
- DML offers the right level of abstraction to the application programmer
- DML is capable of integrating standard specifications.

Desired enhancements towards more sophisticated tools for editing and debugging could even more increase the productivity of developers.

8. REFERENCES

[1] ISO/IEC 10165-4 - ITU-T X.722
Information Technology - Structure of Management Information
Part 4: Guidelines for the Definition of Managed Objects
1993

[2] ITU-T Recommendation Z.100
Specification and Description Language (SDL)
Geneva, 1992

[3] ISO 8807
LOTOS: A Formal Description Technique based on the Temporal Ordering of Observable Behaviour
1987

[4] O. Festor
OSI Managed-Object Development with LOBSTER
Proceedings of
5th IFIP/IEEE International Workshop on Distributed Systems: Operation and Management (DSOM'94)
1994

[5] M. Wittich, M. Pfeiler
A Tool supporting the Management Information Modelling process
IFIP Transactions C-12
Integrated Network Management, III
Elsevier Science Publisher B.V. (North-Holland)
1993

[6] ITU Draft Recommendation M.3100
Generic Network Information Model
1992

[7] ITU G.774
Synchronous Digital Hierarchy (SDH) Management Information Model for the Network Element View
1992

[8] DOMAINS Management Language
Final Deliverable of the ESPRIT Project 5165 DOMAINS
Distributed Open Management Architecture in Networked Systems
April 1993

[9] DOMAINS Management Architecture
Final Deliverable of the ESPRIT Project 5165 DOMAINS
Distributed Open Management Architecture in Networked Systems
April 1993

[10] A. Fischer, M. Herpers, D. Holden, S. Sievert
The DOMAINS Management Language
Integrated Network Management, III
Proceedings of ISIMN Symposium in San Francisco, USA, April 1993
IFIP Transactions, North-Holland
1993

[11] Eike Gegenmantel
Generic Information Structure for SDH Management
International Journal of Network Management, Vol 4, Number 1
March 1994

[12] ISO 8824
Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)
1987

[13] Bertrand Meyer
Object-oriented Software Construction
Prentice Hall International,
1988

9. BIOGRAPHY

The authors work in the Architectures and Systems department at Philips Research Laboratories in Aachen, Germany. Their main focus is directed on network management.

B. Fink received in 1967 a Diploma in Electrical Engineering from Technische Hochschule Aachen, Germany. Her key activities are architectures and computer languages.

H. Dercks graduated in computer science from the Technische Hochschule Aachen, Germany in 1978. He is specialist in systems engineering and compiler development.

P. Besting holds a master's degree and a PhD in Physics from University Bonn. His main interest is application creation and transmission and switching technologies.