Thomas G. Muth

Functional Structures in Networks

Thomas G. Muth

Functional Structures in Networks

AMLn - A Language for Model Driven Development of Telecom Systems

With 191 Figures

Springer

Dipl.-Ing. Thomas G. Muth
Frimans vägen 13
141 60 Huddinge
Sweden
thomas.muth@telia.com
www.amln.se

Library of Congress Control Number: 2004116863

ISBN 3-540-22545-5 Springer Berlin Heidelberg New York

Springer is a part of Springer Science + Business Media

springeronline.com

Printed in Germany

Typesetting: Digital data supplied by editors
Cover-Design: design & production GmbH, Heidelberg
Production: medionet AG, Berlin

Printed on acid-free paper 62/3141 Rw 5 4 3 2 1 0

Foreword

The architecture of "The Worlds Biggest Machine[1]" is of course expressed in the many underlying communications standards; however it is far from explicit nor readily accessible. While on one hand, marketing people are busy looking for their so called "Killer Applications" that will maintain the economic growth of this machine, the engineers are struggling to keep up with the myriad of networks, protocols and standards that interconnect an ever growing number of network services across a rapidly increasing variety of platforms and protocols.

Within the industry, it is commonly accepted that fewer than 10% of engineers working in the field have sufficient knowledge and experience to tackle the pre-study and feasibility phases; that is to say, only this group can process the knowledge and overview of the elusive architecture that allow them to identify the network nodes, network services, protocols and messages that will be affected by adding new network functionalities.

It follows that 90% of engineers are capable of performing the execution phase. This generalization has served as the motivation behind attempts to automate the generation of code for proprietary programming language, a major investment that ultimately failed totally. This long standing objective of moving away from implementing services, one line of code at the time, is however gradually becoming a reality for standardized languages. The software provider I-Logix, for example, has been successful in developing tools that not only generate fully executable Java and C++ code but also perform the critical round tripping.

Mr. Leif Edvinsson, winner of the prestigious Brain of the Year award in 1998, has developed a model to describe the hidden brain power of a company, the *Intellectual Capital* (IC) management model. In the context of the 10%–90% discussed above, these hidden tangible and intangible values describe a company's IC as the sum of the following five elements: 1) intellectual property (IP), 2) knowledge management (KM) systems, 3) the employees' knowledge, 4) the employees' experience, and finally 5) the employees' skills. Note that while knowledge is tangible and transferable, experience and skills are neither. In the case of proprietary systems and languages, tools support and training will be high. However, it does afford some measure of protecting (retaining) the skills and experience established.

---

[1] "The Worlds Biggest Machine" is the title of a white paper, "3G: Building the World's Biggest Machine" by Steve Jones, originally published on the 3G portal web page, http://www.mobiletech-news.com/info/2003/10/04/000100.html.

In many instances, knowledge management systems are at a bare minimum or non-existing, thus increasing the dependency on specialized skills and experience as the company's IP knowledge is held and managed by a very few individuals, that is, the "less than 10%" group. Do you know what your IC is doing?

The reality of today's market place is that being first on the market with a product can mean the difference between success and failure. On one hand the protection of a proprietary programming language quickly becomes costly to maintain and an overall burden. On the other hand, non proprietary languages, third party development tools and "free ware" are slowly but surely taking us away from the paradigm of developing applications one line of code at the time, reducing costs and shortening the time to market. All companies do of course have equal access to these same resources, including the pool of engineers that represents the remaining 90%.

In the context of the 10%–90% discussion above, the question remains "just where is a company's IC vested?" My personal view is that the majority (well over 50%) of a company's intellectual knowledge-based property is vested in the knowledge and experience of the less than 10%.

In response to this conclusion, this book by Thomas Muth, is the key to opening the door to that 10% critical mass of network knowledge that is crucial for the understanding of present and future architecture of the so called "Worlds Biggest Machine". Reduce your risk exposure by anchoring your IC with AMLn.

Montréal
October 2004

Jens J. Larsen, P.Eng
Senior Engineer
Ericsson Research, Canada

Preface

This book is about creating and using models as a means to describe and communicate the purpose and architecture of network systems faster and more accurate.

Today, the architecture of standardized systems is rarely described explicitly. An average protocol or interface specification may comprise several hundreds of pages. For a network system, such as ATM, GSM, ISDN, and others, the number of standard pages produced for its specification easily exceed 3000 pages each. For example, to become an ISDN expert, you have to read (and understand!) over 7000 pages. A complete specification of a network system also comprises a large number of system documents that specify how an implementor intends to realize the standard. Basically, all this information is presented textually, possibly accompanied by some ad-hoc figures.

A development process that is based on this way of describing a system is *document-driven*. It does not take much imagination to see the problems this creates for a network system designer: the facts the designer needs reside in hundreds of large documents that are far from easy to survey. In reality, the designer relies heavily on the few experts who have been engaged in standardization and early system-developing phases, and must repeatingly attend seminars and courses given by them, in order to understand a network system and its evolution. Moreover, it is also practically impossible to keep system documents updated since accurate tracing of requirements is very difficult. The architecture of the system therefore tends to deteriorate, its purposes get veiled in obscurity as time goes by, its evolution slows down, and the introduction of even minor improvements and service enhancements become major design projects.

After being exposed for several decades to the rather intuitive and ad hoc way we describe networks and systems, I decided to write this book in an attempt to make life easier for most network designers, whether you are a manager with technical background, whether you work with standardization, the architecture of implementors' solutions to standards, product design (constructing the software and hardware parts of operator networks), system testing, or whether you are an operator responsible for the configuration and maintenance of a network.

Modeling as a basic principle for producing specifications turns design from being document driven to *model-driven*. As humans, our creative thinking is based on modeling of almost everything we can think of (e.g., an artifact or a sociological system). In the context of this book, however, the only model type that is discussed is the *information model* that describes the purpose, structure, and behavior of a network.

Today we must deal with hundreds of documents and many thousands of pages in order to understand a system. My firm belief is that we would all vastly benefit if standards and implementors' solutions to standards were expressed as models instead. The potential benefits of a model-driven development process are enormous, in particular if models are created in the early phases, preferably already in the standardization phase: instead of interpreting documents, an implementor can build a model of his solution to standards by refining the standard model (using the same modeling technique). The potential for requirement tracing, shorter time-to-market, and many other benefits should be obvious to everybody.

I guess that you already know a lot of today's networks (or at least some aspects of them), so I am not trying to teach you anything about them Once you have read this book, you will find that now you are also the expert on modeling, at least as network modeling is concerned. You will then be the one who defines the requirements on the languages and tools you badly need, instead of being told what to use (by software system modeling experts). When modeling has become the general method in your work, you will also experience the satisfaction of being a system expert yourself.

The basic prerequisite for being able to enter the model driven path for network system development is a suitable *modeling language*, which is the subject of this book and my contribution in this respect. There is only one thing I must ask from you: you have to understand what modeling in general, and network modeling in specific, is about. Until now you have trusted and relied on (software) modeling specialists to provide the design languages and tools you use in your daily design work. This is just too much to ask. These specialist know how to design languages in general and they can design the most sophisticated tools for you. But you cannot expect from them specialist knowledge in your particular knowledge domain (i.e., networking). This is the reason that all you have been receiving are languages and tools for hardware and software system modeling (e.g., UML, the Unified Modelling Language[2]), but not tools for modeling network systems.

Systems modeling has been a rather hot topic for quite a few years. The wide spread use of UML is one result. UML is a standard and therefore perceived by many designers as a modeling language for everything. So, why did I have to develop another? The answer is simply that there is no such thing as "a language for everything" since formal languages are context-dependent. Different languages are needed in different application domains if they are to be useful. Every domain needs to describe its systems in terms of concepts and design rules that are characteristic for that domain. For example, you might be able to describe an aircraft system using the terminology and symbols of the telecom domain, but (obviously) nobody would understand anything. A language such as UML can however play an important role for modeling *system types (component systems) that exist in many*

---

[2] UML has been standardized by OMG in an effort to unify diverse object oriented languages for modeling software systems

other system types, such as software systems and hardware systems (electronic devices). Viewing a network system as consisting of such component systems is something you do first when the standardization phase and the initial phases of an implementor's system development is passed. We may therefore regard UML as a *common-value* language in relation to *added-value* languages, such as the one presented in this book.

To be useful, a common-value language must be completely transparent to the concepts and semantics of any other system domain, i.e., the model cannot tell you anything of the purposes with the added-value system. It only tells you what its software and hardware components are and how to connect them when building an added-value system (in our case, an operator network). An added-value language, on the other hand, is built on the concepts and architectural principles of a specific system domain (e.g., network systems), i.e., on the knowledge of what this type of systems do, what their functions are, and how to combine them into an architecture that is open for its evolution over a long time. In other words, an added-value language is based on, and preserves the *intellectual capital* of a system domain. Since mapping an added-value model on a common-value model means that the semantics and architecture of the added-value system gets completely (and deliberately) lost, creating and maintaining added-value models is extremely important to any system-developing enterprise.

The added-value modeling language that is presented in this book is called AMLn (*Abstract Modeling Language, network view*). To my knowledge, it is the first and, so far, the only attempt to create a modeling language for network systems. The background to AMLn is the enormous complexity and rapid evolution of today's network systems, caused by the introduction of new technologies in electronics, software, and transmission media over the last 25 years. As a result, new network systems appear in an ever increasing pace. One would expect that, over these years, well-proven and efficient architectural principles have been identified and commonly applied. On the contrary, only a single attempt in that direction has been made: the OSI RM (Open Systems Interconnection Reference Model), published in 1984. This generic model exhibits such a set of general architectural principles. In spite of the fact that the OSI RM was only a first attempt in the right direction and was constrained to experiences from a small class of pre-standard datacom networks (e.g., IBM's SNA, DEC's DNA, DARPA's Arpanet), the industry took it to its heart with 100% consensus, and with great enthusiasm. In retrospect, we can conclude that this was more an effect of a fear of losing control of the evolution than of a serious evaluation of the quality and applicability of the OSI RM.

Unfortunately, the industry still has not pursued such an evaluation. Even though the idea of "OSI networks" failed completely when confronted with actual needs in public networks and more sound architectures (such as the Internet), the industry continues to align the description of today's network systems to the OSI RM, particularly in standards. Since a competent network architect does design architectures for optimal maintaineability and openness to new technologies and

requirements in mind, the strange situation exists that many standards continue to describe sound architectures in terms of a generic architecture (i.e., the OSI RM) that is not reflected by any network system in use today. Those standards are next to unintelligible.

I was as enthusiastic as anybody else over the OSI RM at the time. However, when entering the field of network systems design later, I gained a deeper knowledge of how networks systems were actually built and what the real architectural needs were. This knowledge and experience has been transformed into AMLn. In this endeavor, I had to deal with two problems:

1. Finding the minimum set of concepts that could be the corner stones for the language. For a system domain that has been developing for over 100 years, those concepts should have been available for a long time. Except for concepts defined by OSI (which to a large extent are insufficient for modelling purposes), this is not so, however. The ITU–T (a leading standardization body for telecommunication standards) tried for many years by publishing a steadily expanding list of thousands of terms. It gave it up some years ago when it found that, in reality, every standard defined its own concept definitions. Thus, this issue became a major research topic in developing AMLn.
2. Identifying the architectural principles that result in a stable system structure over a long time, and a model that is easy to understand, manage and relate to common-value models.

The specification of a system type must cover two aspects: its *behavior* and *structure*. Behavior specification is a topic that can be covered quite acceptably by most common-value languages. Structure cannot, however, since it requires the identification of efficient architectural principles of the actual domain. As network systems are concerned, a few principles are common knowledge, e.g., separating access network problems from core network problems. Some new principles (e.g. the "layer" concept) were defined by the OSI model. Over the last two decades a number of new principles have emerged (e.g., the concepts of "horizontal networks" and "logical networks"), which cannot be defined in terms of common knowledge and the OSI model.

In AMLn, all essential architectural principles (old, useful ones and new ones) are built-in. The general principle applied for managing complex system is "separation of concerns." AMLn is based on two main concerns:

- Separating *functionality* from *connectivity*. These subjects are covered in Chaps. 2 and 3 respectively.
- Separate the system that offers services to system users (the " managed or *traffic system*") from the system that manages that system (the "*management system*"). How these systems are related is described in Chap. 5, that also discusses existing management systems (TMN, SNMP, CORBA and OMAP) through AMLn models.

Whenever you separate concerns, you must also define how to *integrate* the parts, since a system consists of related parts. Chapter 4 discusses this subject as regards the separation of functionality from connectivity. The chapter also considers that the specification of a network system is always created by breaking down its structure step by step. The structure of an AMLn model that results from this is characterized as its *network-level* structure.

Chapter 6 presents some case studies where AMLn has been applied as an analysis tool on well known network systems.

An overview of AMLn is given in Chap. 1. A brief introduction to modeling in general is also included, as well as the definition of the *logical dimensions* in which an AMLn model may be viewed. Logical dimensions are conceptual tools for creating and viewing the information structure of an AMLn model. They also define the structure of an AMLn model data base.

This book is a successor to my previous book (see Muth (2001)), which describes two other views of AML: AMLs (*AML, service view*) and AMLp (*AML, protocol view*). Both these views deal with behavior specification and are applicable when one wants to refine an AMLn model for simulation, or for generating code directly from such a model. A brief overview of AMLs and AMLp is given in Appendix D at the end.

During the time I have been struggling with this book, many people have encouraged me and given me valuable feedback. I would especially like to thank Per Dahl, Anders Eriksson, Lennart Holm, Jens Larsen, and Anders Olsson.

Stockholm
October 2004

Thomas Muth

Contents

1 Introduction to Network System Modeling

2 Layer Structures

3 Node Structures

4 Modeling Vertical and Horizontal Partitions

5 Management and Traffic Systems

6 Applying AMLn

1 Introduction to Network System Modeling

1.1 Systems Modeling in General

This book presents AMLn (**Abstract Modeling Language, network view**). AMLn is a proposed language for modeling **network systems.** We regard as "network system" any system that exhibits a structure of nodes and provides connectivity services for humans, machines, and different types of application systems. Network systems are commonly defined by international standards (AMLn is, however, not restricted to standardized systems only). Examples of standardized network systems are the Internet, UMTS, PSTN/ISDN, SS7, GSM, GPRS, ATM, SDH, and diverse LAN concepts (see Appendix A for definitions of acronyms).

In this chapter we provide a general background to modeling. When we want to describe something, whether it is an artifact or something more abstract, we can do it verbally, using natural language text in documents (perhaps with some supporting pictures) or we can build a model of it. A model may be a miniature of an artifact, or it may be an information model (the only interpretation of modeling that concerns us in this book). In an information model, parts and relations between parts of the thing we model are represented by information elements.

An information model is built by using a modeling language. A modeling language is designed so that a model in that language can be understood not just by a human interpreter, but also by an interpreter that runs in a computer system. Thus, a modeling language is a formal language with special qualities for human interpretation of models that are constructed in that language. For example:

- It includes concepts and terms that are familiar to designers and users within a particular system domain (which may be networks, power plants, missiles, software systems, etc.).
- It includes graphical representations of important basic language concepts.
- It allows names and acronyms to be used instead of neutral identifiers for entities in the model.

Since specifying network systems is still document- and not model-driven, we need to emphasize the enormous power of modeling in enhancing creativity and improving communication between people, compared to providing text in documents. *Modeling* is the creative thinking that you do when trying to find a solution to a problem. *Documenting* is what you do when you describe your solution. Thus, if you create a model of the system instead of just documenting it, not only do you

improve your own thinking, *the final model also becomes the documentation* (provided that you use a known modeling language and some discipline). Another important side effect of a modeling approach is the possibility to add information to the model that explains *which problem* a certain feature solves, and *why* the particular solution has been chosen in favor of already known solutions to the same problem (something you seldom find in traditional system specification and description documents, in particular not in network standards).

Every modeling language uses the same basic paradigm for describing the *structure* of a system, which is that "a system consists of related parts." Considering what we know about the complexity of most systems today, this view seems to be too simplified to be useful (especially since the *behavior* of systems is not included in the definition). It is, however, of value for something we may call a *universal abstraction*.

This abstraction has been the paradigm for modeling languages that produce things such as *semantic diagrams*, *conceptual models*, *entity-relation diagram* (ER) or *entity-relation-attribute diagrams* (ERA). Whatever name has been used, these languages all support modeling on the level of universal abstraction:

- Semantic diagrams were the first to be used, in particular for modeling concepts (e.g., the concept of "religion") and conceptual systems (e.g., the system of numbers). The purpose with this type of models is primarily to explain the meaning with concepts for human readers (the model may, for example, model the statement "a car has four wheels" as the *relation* "has" between the concepts "car" and "wheel", also adding the *attribute* "four" to the relation "has").
- With the advent of electronic computing, the things one needed to model was a part of the physical world (e.g., the business control system in an enterprise) that had to be supported by the computer. This part was modeled as a conceptual model (or equivalent: an entity-relation diagram). The goal with such a model was to be able to first understand the actual part of the reality, and then translate that model into a data model (i.e., a data structure) that could be stored in the computer. The things that appear in such models are now no longer any type of concept (as in a semantic diagram), but representations of *entities* of the physical world. Some type of relations seemed to recur in models of all system domains, such as is_part_of and belongs_to_the_class, and are therefore often defined as parts of the modeling language itself.
- The next step in the development of modeling languages took place when people started to use modeling not just to define data structures in computers, but for modeling the computer system itself. Some early modeling languages tried to cover all aspects of a computer system in a model. With the enormous increase of software in systems in general, however, successful modeling languages have focused on modeling the software part of it.

During the 1980s, the first step to include behavior in models was introduced. It became popular to regard the entities in models as *objects* and the relations as *object relations*. An object is an entity that hides its design behind an *interface*, and

an entity which can execute *operations* upon requests from other objects. The structure of the system can now be described by interfaces between objects, and the behavior as interactions between objects, requesting operations and responding to requests (sometimes described as "protocols" between objects).

Object oriented (OO) modeling languages in use today have, in reality, only been developed and used for software system modelling. A well known language of this kind is the Unified Modeling Language (UML).[1] The fact that such languages are constrained to systems that have a single object type (which is a piece of software or data), makes it possible to pre-define a larger number of relations than just is_part_of and belongs_to_the_class. For example, software objects exist both as classes, types and as instances, where an instance is related to a type by an is_an_instance_of relation. Most operating systems also offer connectivity relations between functional elements, which can be modeled as an is_connected_to relation, etc.

OO modeling languages were preceded by OO software design languages, such as Smalltalk, Ada, Eiffel, C++, and Java. Since you can express almost the same thing in these languages as you do in UML and similar modeling languages, the latter tend to be merely graphical versions of OO design languages. The extra value they give, besides a graphical notation and concepts for the universal abstraction level, is when diverse packaging concepts are added that make the model useful in managing the overall complexity of a large software system.

When modeling languages have the ambition to be useful beyond the level of universal abstraction and/or include behavior modeling, the complexity of the models normally require the language to define some kind of *views*. One distinguishing aspect between languages is therefore their choice of views. Some languages that are called "modeling language" offer no views at all, and should therefore be called "drawing languages" instead. A view is the picture you get of a system model when looking at it from a particular "angle." For example, a mechanical design exists in 3-dimensional space and the model of it is normally presented in separate drawings for a number of cuts in that space. In the case of more logical (or abstract) types of systems, such as networks, computers, or software systems, the selection of views is not that obvious.

Basically, the selection of views depends on the type of system that the modeling language targets. For example, UML (a language designed for software-system modeling) defines a large number of views, called "diagrams" (use-case diagram, class diagram, state diagram, activity diagram, object diagram, etc.). These are (hopefully) relevant and of interest to software system designers. A language for computer system modeling may define additional views that show other properties of the system, such as its system bus and connected hardware, memory organization, power supply subsystem, and mechanical design, including cabling.

---

[1] See The UML Reference Manual. Object Management Group, Framingham, MA. http:// www.omg.org/. Cited 1998

Note however that there is no consensus between similar languages regarding which views should be supported for a particular system domain (the choice of views seems to be a means for competition between languages instead).

1.2 Added-Value versus Common-Value Languages

1.2.1 General

The previous chapter discussed how far you can get on the level of universal abstraction and/or by using *common-value* languages (e.g., languages for software and/or hardware system modeling). For productive modeling of systems within a particular system domain (e.g., networks) we must define more specialized languages, i.e., modeling languages that are based on the concepts and relations that are unique to a domain (such as the relation "connection" and the entity "layer" used in network modeling, or the entity "exhauster" in modeling cars). This is so because we neither want, should, nor can model the system in terms of its software and hardware components before it has to be implemented. We will refer to such modeling languages as *added-value* languages.

It has been advocated (e.g., by UML proponents, see Chap. 1.2.2) that UML (a common-value language) should also be used for modeling systems in domains other than software. The fact that this is theoretically possible does not, however, imply that it is realistic and meaningful. Let's take a look at the implications of such an approach. Figure 1.1 shows the general paradigm for all modeling languages.

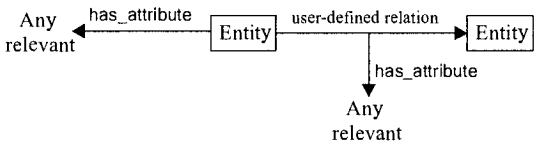

Fig. 1.1 The model for all modeling languages

The model describes the language as defining *entities* that may have any kind of user-defined *relations*, some of which may be *attributes* to entities (attributes are things that have a value range and no other relation to entities in the actual model). Some languages allow relations to have attributes as well. Common-value languages normally define only two symbols, one for relations and attributes and one for entities. Some languages (e.g., the language component in AMLs for modeling resources) do not even use any symbol for entities, i.e., entities are just text strings. Since a common-value language cannot define symbols for entities and relations of added-value systems, most common-value languages realize that they must define a relation that tells which kind of entity is meant. This is the belongs_to_the_class

relation mentioned previously. Looking for similarities cross many system domains reveals that another common aspect is to describe that something is part of something else. Consequently the relation is_part_of is normally included in common-value languages. For example, UML includes both these types of general relations.

Let's now use UML notations (as an example of a common-value language) for creating a very small part of an added-value model in the network system domain. This model (see Fig. 1.2) tells:

> *Two nodes, N1 and N2, are interconnected by a route, R1. N1 includes a layer element L1. N2 includes a layer element L2. L1 and L2 communicate according to the protocol P1.*

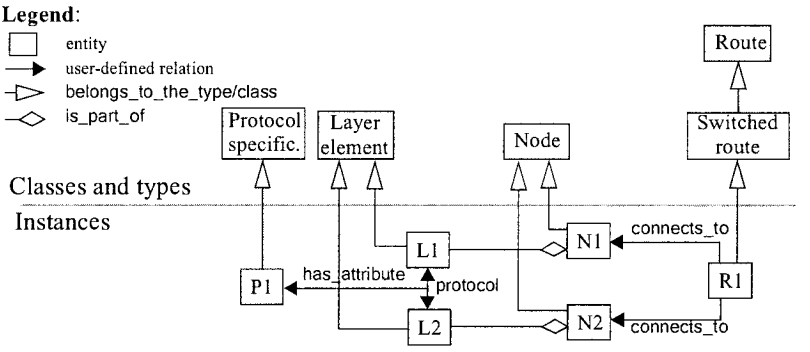

Fig. 1.2 An entity-relation diagram of a network system statement

Using a common-value language implies that added-value concepts (here **node**, **route**, **layer element**, **protocol**, **protocol specification** and **connects_to**) must be defined by the modeller by including specific type and class entities, and by textual annotations of user-defined relations. The model also demonstrates that the meaning of the statement we model is not given by concepts defined by the common-value language, but (at best) by the added-value model. In other words, *the meaning with the model is (to a part) to model meaning.*

The fact that common-value languages use very few symbols may make the impression that they are easy to use and give intelligible models. The model in Fig. 1.2 clearly shows that this is a very deceptive appearance. It does not take too much imagination to see that modeling a larger part of a network system, with all its domain-specific entities, relations, and attributes in a common-value language would create a model that would be extremely complex and hard to understand (and therefore useless). The model would be cluttered with class and type objects, class and part relations, and user-defined annotations that exist only to define what type of entity, relation and attribute is meant. This is definitely something we do not need when modeling a system that is complex in itself. Therefore let's look at how the same statement is modeled in AMLn (see Fig. 1.3).

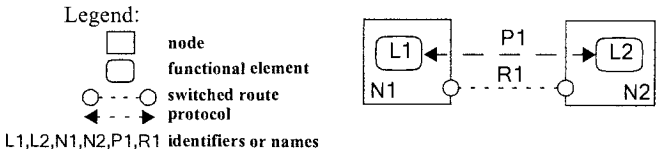

Fig. 1.3 The AMLn model that corresponds to Fig. 1.2

We leave to readers who work in the networking industry to judge which of the models in Figs. 1.2 and 1.3 they prefer. Here we will only summarize the main differences:

- All entity symbols are domain specific.
- All relation symbols are domain specific, i.e., no user-defined relations are allowed. Therefore, the modeller need not add any text or code to explain what is meant with a particular relation. The excess of textual annotations that characterizes models in common-value languages is thereby avoided.
- The need for the general relation is_part_of is eliminated by graphically including entities that are part of another entity in the latter.
- The need for the general relation belongs_to_the_type/class is eliminated by using domain-specific graphical symbols for entities and relations.

Obviously, an added-value modeling language will have to define more graphical symbols than a common-value language, which could be an argument against it. For example, in the very early proposal for AMLn, the author did use an excess of network-specific symbols, in order to allow for models with high semantic precision. This is counterproductive, however, since a well known fact is that humans have a short memory span that cannot cope with more than something in the range of 5–9 concepts (in this case: symbols) at the same time. For most system domains, this is a tough challenge to the language designer. For the case of AMLn, the author has solved this problem as follows:

1. Carefully analyse and select the minimum set of concepts and relations that are used in all network systems.
2. A specialization that cannot be included in the range of symbols can always be defined by attributes of an entity or a relation.
3. Define a set of model types ("views"), each set using only a small part of the complete set of symbols.
4. More important, most entity types are not defined by symbols of their own, but indirectly by the type of relation they have. Figure 1.4 gives some examples of this approach. Five entity types are used here, relying on only two entity symbols (AMLn defines only two entity symbols: a squared shape, denoting the general entity type **node**, and a round shape, denoting the general entity type **functional element**, which is any entity that shows a pattern of behavior).