A fresh view on model-based systems engineering: The processing system paradigm

Thomas Muth
AB Vascaia
Frimans väg 13, S-141 60 Huddinge,
Sweden

Dominikus Herzberg
Ericsson Eurolab Deutschland GmbH
Ericssson Allee 1, 52134 Herzogenrath,
Germany

Jens Larsen
Ericsson Research Canada
8400 Decarie Blvd, Town of Mont Royal
OC, H4P2N2 Montreol,
Canada

Abstract. Model-based systems engineering and graphical notations have an enormous potential for increasing design productivity, system quality and lifetime by shifting the bulk of design efforts to early phases. In spite of that this is hardly questioned, the shift towards model-based approaches has not come to a break through, as we are experiencing in software engineering. It is believed that a major reason is lack of a common system view that can act as a framework for developing modelling languages and methods for a broad community of system engineers.

This paper suggests such a framework. It identifies the systems of concern as a *processing system* that consist of a *process control system* and a *resource system*, and applies two related views on processing systems: A **functional** view and a **solution** view. The framework has been successfully tested on telecommunication systems and networks for some years. It is believed that it holds for many other system domains as well.

INTRODUCTION

Systems engineering activities are still very much paper-driven. The shift towards model-based systems engineering has not come to a break through, although model-based engineering is becoming more and more popular in the software engineering community. Although languages for modelling software systems, like the UML-the Unified Modelling Language, see (OMG 1999)-do not solve the problem of modelling systems in general, the power of formal and graphical notations for describing and specifying models is commonly accepted. Astonishingly, for systems engineering model-based engineering is perceived as a new paradigm; according to (Fisher 1998) it marks a shift from a paper-driven approach towards functional models, using a computer-aided systems engineering (CASE) tool.

However, no "UML" for systems engineering has been developed so far. This comes to no surprise, because systems engineering applies to a wide range of domains and applications; no modelling language will satisfy all needs of all domains. Part of the solution to create suitable modelling methods beyond those that are used for modelling software or hardware systems are:

- To identify key system characteristics that **unite** as many system domains as possible (without being too general).
- Considering the complexity of total systems, to find an efficient way of **separating concerns** (or characteristics) of systems.

From a **functional** point of view, a simple unifying paradigm for many system domains is to view a system as an entity that produces some kind of utility for users, with a certain degree of autonomous processing. Such systems are called *processing systems*. The processing system concept presented is the result of an in-depth study of telecommunication systems and abstracts the insights gained from these investigations. Consequently, this concept is first presented in general terms and then referred to the telecommunication context; it should help the reader understand the power and general applicability of the approach presented.

From a **solution** point of view, most systems of concern can be distributed systems. The unifying paradigm for distributed systems that is presented next in this paper is the *sysnet* concept. The term *sysnet* denotes that a distributed system has a network structure, and that a whole network is a system, seen from the environment.

Some conclusions regarding modelling and specification languages and methods can be drawn from these paradigms, which is the final part of this paper. This part is based on a modelling language, called *Abstract systems Modelling Language*, AML, which is also briefly described

THE FUNCTIONAL VIEW

The general model. Figure 1 depicts a general, functional model for processing systems.

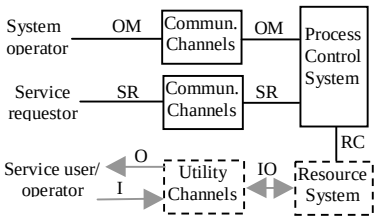

Figure 1. The processing system model

The model distinguishes parts of processing systems that are unique for the type of system (dotted line boxes) from those parts that are not, at least not in the sense that they require specific modelling techniques. Since it must be possible to apply systems modelling without knowing anything about the components of the implemented system, the entities of the model must be discussed and specified only through the interfaces that the model identifies and specifies (in this model called OM for operation and maintenance, SR for service request, IO for input and output and RC for resource system control). This has three major implications: A system model describes **interfaces**; an **entity** of the model is described by how it reacts to interaction events in all its interfaces; interfaces must be described in an **abstract language** that supports the translation of abstract interface specifications to interface descriptions in suitable implementation languages.

The model makes a distinction between three roles that external entities play: *System operator*, *service requestor* and *service user*. That system operator and users play different roles is a reasonable assumption for most processing systems. There are also at least three good reasons for separating the roles of service users and service requestors: These roles are often played by different entities in the environment; user and requestor use completely different types of channels (with the exception of telecom systems); communication channels are not necessarily unique for the type of processing system, while utility channels are.

The model defines three channel types that are used to connect entities in the environment to the entities of the processing system: One *utility channel* and two *communication channels.* Communication channels are separated from utility channels, since they might be very different. Communication channels are used for information transport. The two channels in the model are both adapted for data communication, since they are used by service requestors and operators to send commands to and get responses from the processing system. The channels are provided by some communication system that is not necessarily part of the processing system.

Utility channels are adapted to the actual type of utility flow, which can be artifacts, matter, energy (and even information, in case the processing system is a telecom system). E.g., a fabric dying processing system provides dyed fabric as output (O), provided that it gets fabric, dying solution and heating as input (I).

Besides the channels, the entities of the model consists of a *resource system* and a *process control system* (PCS). A resource system handles interfaces of type IO and RC. A resource system consists of all resources that take part in processing the input to provide a utility as output. How to design a resource system, therefore, requires domain unique competence, and lies outside what can be described by the general model. However, once a resource system is designed and understood, it can be described in a general entity-relation-attribute (ERA) type of modelling language. Such a model is a *resource system model*.

The purpose of a resource system model is to describe the structure and all those aspects of the resource system, which have to be controlled by the PCS. In some way this information will be implemented in the functionality of the PCS, which is why ***resource system modelling*** **is the very prerequisite for modelling the PCS.** The PCS controls the resource system over resource control interfaces (RC). These interfaces are implemented by *sensors* and *effectors* that logically are regarded as parts of different types of resources.

The general model also defines a **high level control structure.** Most PCSs operate with some degree of autonomy, but the system operators have the overall control responsibility. A system operator can control semipermanent aspects of the resource system over the OM-interface. Service requestors control service provisioning over the SR-interface. The PCS controls the resource system over the RC-interface in two respects: As a result of service requests, to lead the resource system through a sequence of *temporary states* so that the output becomes what the user wants; as a result of operator commands regarding e.g. configuration changes, to set the resource system to *semi-permanent states*.

For a more in-depth discussion of the processing system concept in the telecommunication systems domain, see (Muth 2001). In (Herzberg and Marburger 2000) this concept is given a more general and formal treatment, and its relation to the concept "real-time system" is analysed.

Process control system structure. For the PCS to be able to perform its control tasks, it must have knowledge about the resource system. It must also have knowledge about users, e.g. the identity of service requestors, which service a particular service requestor is allowed to use, which utility channel shall serve the service user, etc. This knowledge is modelled in a *user model*, which in some way is implemented in the PCS as semipermanent user information.

The PCS has three main functions: To translate service requests to actions in the RC-interface with a temporary effect on the resource system; to translate commands from system operator (regarding the resource system) to actions in the RC-interface with a semipermanent effect on the resource system; as a

result of commands from operators, perform semi-permanent changes to user information.

As long as distribution is not an issue, the internal structure of a PCS is affected mainly by the complexity of the *resource system model*. A complex resource system is best described in several levels of abstractions, called an *abstraction hierarchy*. The levels of this hierarchy will be reflected by the internal structure of the PCS, the *PCS-control structure*, see Figur 2.

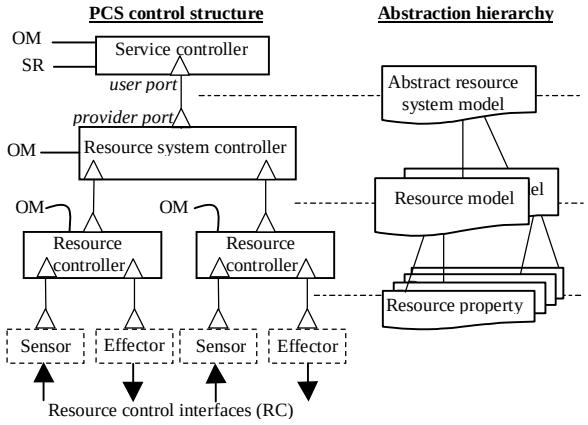

Figure 2. The control structure of a PCS

All entities in the PCS control structure are functional entities that communicate over *user-provider* oriented communication interfaces, called *service relations*, by means of *abstract service primitives* (ASP). In a service relation, the *user-port* (a triangle turned inwards) is connected to the *service-port* (a triangle turned outwards) of another party, who is the one that provides the service in the interface. RC-interfaces are not communication interfaces. *Sensors* and *effectors* should therefore be modelled as part of the resources they interface.

The abstraction hierarchy in Figure 2 is quite general. It recommends the resource system model to be both *fragmented* and *abstracted*. It should consist of one fragment for each significant resource, a *resource model*. This fragment gives an abstracted image of the actual resource, constrained to those structural, quantitative and qualitative properties that are of interest to the *resource system controller* to monitor and affect.

Resource models will not define relations that exist between resources (e.g., the fact that a particular transmission line is connected to a particular switch, or that a particular channel in a trunk is part of a network-wide, temporary connection). This type of information is of interest to the *service controller* (the details on the resource controller level are not). This information is therefore specified in an *abstract resource system model*. Depending e.g. on the complexity of the resource system, the abstraction hierarchy could include more than one level of such models.

An important purpose of the models in the abstraction hierarchy is that they implicitly define the functional complexity of entities in the PCS-control structure: Each entity operates between two abstraction levels, defined by the resource model in its upper interface (where its *user(-s)* are) and the models on its lower interfaces (where its *providers* are). All these models concern the entity, and are implemented inside the entity, e.g. as data models. The role of the entity is to mapp interface interactions between these models. The difference in abstraction level between the user and provider side of an entity therefore (implicitly) defines the functional complexity of the entity and the processing power it needs. Consequently, sensors and effectors belong to the lowest level on this scale, the resource system controller to the highest. There is also a trade-off between functional complexity of entities and the responsiveness of the PCS, that is reflected by the number of levels that are defined in the abstraction hierarchy.

In summary, to carefully analyse and design the abstraction hierarchy (i.e. to produce and manage the resource system model) is a very important aspect on PCS-design. It is also a necessary prerequisite for behaviour description of all entitities in the PCS-structure, since the implementation of resource system model inside entitites define the state of the resource system.

Telecom systems as processing systems. The general processing system model has to be specialized for the particular systems domain at hand. Figure 3 shows its specialization for telecom systems.

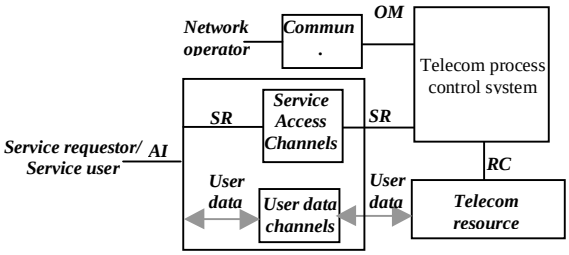

Figure 3. The telecom processing system

The unique utility provided in this case is connection-less (CNL) or connection-oriented (CON) transport of information or data between service users (i.e., the *communication channels* that other types of processing systems need). CON-services are characterized by that a channel that is adapted to the type of *user data* that service users want to exchange, is (often temporarily) made available to them. Another unique feature is that service requestor and service user is mostly the same entity, and that, since both types of user interfaces implies data transport, they often are combined in channels over a common access interface (AI).

Due to the complexity of the telecom resource system, a *telecom process control system* often comprises at least two levels of abstractions of the telecom network: *Connection control* and *call control*, see Figure 4.

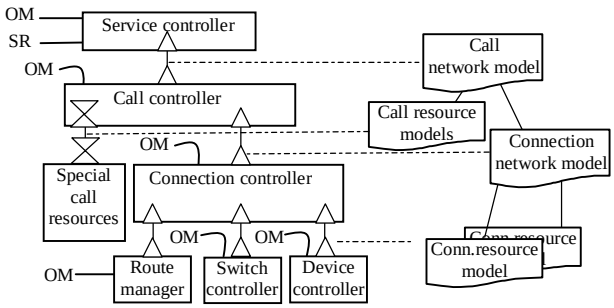

Figure 4. The telecom process control system

A *connection* is a pure "bit-pipe" that is created in the resource system by connecting channels in trunks through switches in the resource system. Connections are controlled by the *connection controller*. By connecting *special call resources* (e.g. voice encoders) in the resource system over connections, channels between service users are created that are adapted to transporting a specific type of information. Such channels are *calls*. Calls are controlled by the *call controller*.

Entity modelling. The function of an entity of the PCS control structure depends on several resource models. There is one model on the side where the user(-s) are, as well as one or several on the side where its providers are. All these models are *service resource models*. Since the function of the entity is to map interaction events between interfaces, the entity must implement all models inside, and know how they are related. The sum of all models and their relations is the *object resource model*, see Figure 5.

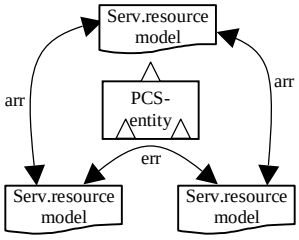

Figure 5. The object resource model

To specify object resource models, once the service resource models are created, is a relativily simple task. Relations between resource models on different abstraction levels are called *abstraction resource relations* (arr). Relations between resource models on the same abstraction level are called *external resource relations* (err). From an analysis viewpoint, it is important that these relations are separated in the object resource models, since requirements on the PCS often are related to one particular service resource model.

THE SOLUTION VIEW

Introduction. A processing system can be a single physical entity (a *physical node*), for example a PC that controls some (local or remote) resource, or a switching node in a telecom network. A processing system can however also be any kind of network (a *physical network*) of such entities, and any partition of a physical network can be regarded as a physical node. The term *sysnet* denotes any kind of system that has such properties or potentials. Processing systems can be sysnets, and so can information systems, data base systems etc. Thus, the concept *sysnet* has a wider applicability than just processing systems.

As soon as physical distribution becomes an issue for a system, there are a number of additional design decisions that have to be made, and a number of functions that have to be added to the functional model in order to be able to implement the system as a sysnet (i.e., as a physical network).

Functional space. In a sysnet, entities are not just functional entities. They also have a "location" with respect to the **physical structure** of the sysnet, with respect to some network-wide structure of **layers** for different types of functions, and with respect to processing nodes or **platforms** (which could be distributed systems themselves). By an analogy to how the location of concrete objects are identified in spatial space, one can define a *functional space*, consisting of three *logical dimensions*, see Figure 6.

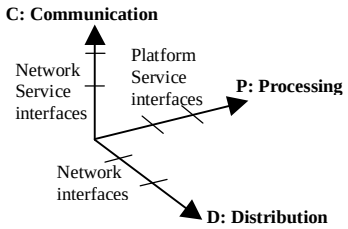

Figure 6. Functional space (DCP)

A sysnet **exists in functional space** (also called the DCP-space). This means that the structure of the system is defined with respect to three different types of interfaces: *Network interfaces* in the *distribution dimension* (D), *network service interfaces* in the *communication dimension* (C) and *platform service interfaces* in the *processing dimension* (P). Depending on in which type of interface(-s) an entity in functional space is viewed, it has different names and implications, as to which functions it represents. A *physical node*, see Figure 7, is an entity that is defined in the D-dimension (i.e., it exists in one or between several network interfaces), and it hides both the C- and P-dimensions inside (i.e., network service interfaces and platform service interfaces).

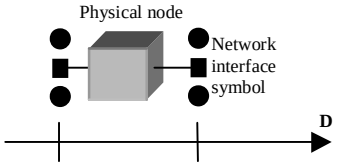

Figure 7. A physical node in functional space

Network interfaces relate physical nodes to each other, thereby creating models of *physical networks.*

A physical node has an internal structure of entities that exist between network service interfaces, and that communicate in the D-dimension by means of *message protocols*, see Figure 8.

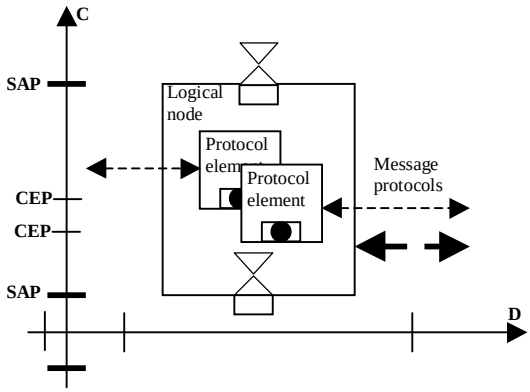

Figure 8. Logical nodes and protocol elements in the DC-plane

There are two types of network service interfaces to consider: *Service access points* (SAP) and *connection (*or *association) endpoints* (CEP). Both types of interfaces are of the user-provider type and offer some type of message transfer service (i.e., sending and reception of data). A SAP is a **permanent**, addressable interface in functional space that connects a *logical node* to a message transfer service. SAPs were first defined by the OSI reference model. Message protocols relate logical nodes to each other, thereby creating models of *logical networks.* Note that *layering* may be applied in the C-dimension, which means that the sysnet may define message transfer services on several layers. A logical network is always confined to a single layer.

Logical nodes can be built by smaller entitites, called *protocol elements*, that implement the entities of the functional model of the PCS. The logical node protocol has to define how protocol elements are associated to each other over (normally) **temporary** CEPs, and how messages to different protocol elements are separated. These mechanisms are parts of the logical node protocol. The remaining parts are message protocols between protocol elements that relate protocol elements to each other, thereby creating models of *protocol networks.* The distinction between logical nodes and protocol elements can be recursively applied on protocol elements. Its purpose is to allow the specification of less complex network functions than logical networks, and to allow concurrently operating protocol networks to exist in a sysnet.

Every physical node must include processing capabilities, which is modelled as a *node platform* (which might be a distributed, layered system itself). The platform creates the characteristics of network functions (response time, transfer delays, reliability, operability, etc.). It supports communication between logical elements inside the physical node (logical nodes, protocol elements or smaller parts), primarily over SAPs and CEPs. The platform does however not send or receive any messages to and from the environment of the physical node. Logical entities in a physical node constitute **requirements** on *network applications* that use platform service interfaces, see Figure 9.

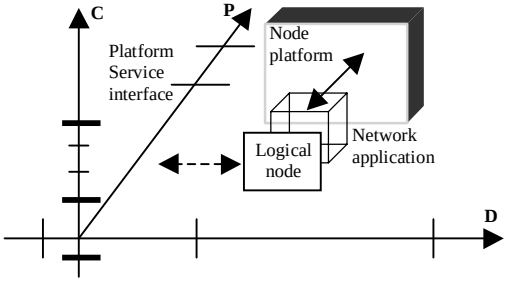

Figure 9. Applications in the P-dimension

Integration hierarchy. Functional models (two types: service networks and abstract networks), protocol networks, logical networks and physical networks are all collaboration types of models that represent different abstractions and parts of real networks or systems. Each model type specifies some particular property, without any overlap to other model types. A complete integration of all models over the five levels of model types defines a physical network. The model types are chosen so that they can be successively aggregated and integrated in each other, by mechanisms that can be standardized.

These mechanisms are generally called *integration mechanisms*. With the exception of service networks, each model type defines one type of integration mechanism:

- In order to support communication between service objects, service networks have to be integrated into abstract networks. The integration mechanism that is added by an abstract network is *object access and management* functions.
- Abstract networks alone cannot operate in a network context, unless they are aggregated in protocol networks (normally one abstract machine per protocol element). The integration mechanism that is added by the protocol network is a *message handling* function that translates an *abstract protocol* into a *message protocol.*
- Protocol elements cannot be addressed unless they are aggregated into logical nodes. The integration mechanism that is added by the logical network is an *association handling* function, that allows temporary protocol elements to be addressed individually over the network.
- Finally, logical nodes cannot be addressed unless they are aggregated into physical nodes. The integration mechanim that is added by a physical node is a *node platform*

that allows logical nodes inside physical nodes to communicate, and that creates the characteristics of network functions.

Together these model types, related through aggregation and integration, constitute an information model structure for sysnet models that is called the *integration hierarchy*.

LANGUAGES AND METHODS

General aspects. Languages and methods that are needed for modelling processing system, both in the functional and in the solution view, are summarized in three groups of method components in Figure 10.

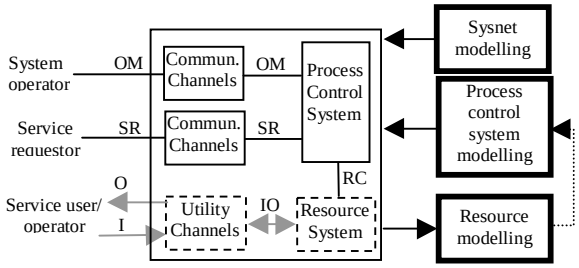

Figure 10. Method components

The resource system model is the prerequisite for producing a functional model of the process control system. It affects the control structure of the PCS, and it is the basis for specifying the interfaces and the behaviour of interfaces and entities in the structure. A method for *resource system modelling* is therefore a core element for modelling processing systems. Most ERA-type of languages can be used for resource system modelling.

Process control system modelling deals with producing a functional model of the process control system. This method component should support:

- The creation of a *PCS control structure* where all resource models are allocated to interfaces in that structure, and where *err*- and *arr*-relations are defined.
- The specification of *interfaces* in the structure. Interfaces in a functional structure should be expressed in terms of *operations* that the controlled party (the provider) offers to the controlling party (the user). Operations must be related to entities, relations and attributes of the service resource model of the interface, since the purpose with performing an operation is to change the state of the resource system, or to inform the controlling party about resource system states.
- *Structural refinement* of the process control structure, in order to create structural solutions to the main objects (service controller, resource system controller, etc.).
- The description of *behaviour* of interfaces and entities in the control structure. Most process control systems are systems that react on real-time interactions in the OM, SR and RC-interfaces. Thus, most PCSs of interest are real-time systems A behaviour of an entity in a real-time environment is best described as sequences of interactions between states of the entity, which implies that (from a behavioural point of view) entities are regarded as **state machines**.

Sysnet modelling is a method component of its own. This component must support the following modelling tasks:

- Specification of protocol networks and *message protocols*. The interfaces that are defined in the functional model are abstract interfaces, since they do not consider the fact that entities may be physically distributed. The method must therefore support the translation of a behaviour described in terms of operation interactions (an *abstract protocol*) to a behaviour that relies on message interchange (a *message protocol*).
- Specification of logical networks and association handling functions.
- Specification of physical networks.
- Specification of aggregation relations between model types in the integration hierarchy.

The Abstract systems Modelling Language, AML. AML is a modelling language that has been developed with the processing system paradigm in mind. AML includes its own modelling language for resource modelling. The support for modelling the functional structure of process control systems consists of two types of collaboration models, *service network* and *abstract network*.

Service network is the primary structural model, to which basic interface and behaviour specifications are associated. All entities are *service objects* (a kind of abstract objects). A service object **provides** a single *primitive service*, but it can have any number of interfaces in which it **uses** primitive services that are provided by other service objects. The extent of a primitive service is defined by the corresponding service resource model. Depending on the configuration of *service ports*, there are three basic object classes in a service network: *Virtual object*, *agent* and *actor*, see Figure 11. These concepts were originally defined by (Booch 1983).

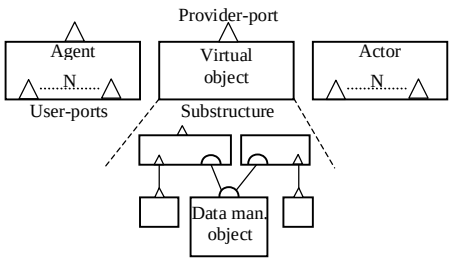

Figure 11. Object classes in AML

Any service object can be recursivly refined by a *substructure*, as for the virtual object in Figure 11. Substructuring creates hierarchical structured service networks, where service objects on higher levels may represent functions that are network-wide, while service objects on lower levels can represent e.g. software objects.

A PCS normally handles many different functions, e.g. functions related to requests from service users and from the system operator. In most PCSs, users may also request a number of different services that use different parts of the resource system. Each major function will be described in a service network of its own. A component of the implementation of a system will normally comprise many service objects, from the same or from different service networks. There is therefore a need to define an *abstract machine* as an entity that can represent such an aggregation of service objects, and that is equipped with a special function, *object handling*, that allows a service object in one abstract machine to access a service object in another. An example of standardized object handling is the CORBA-platform, see e.g. (Orfali 1996).

How service objects are allocated to abstract machines is the designers decision. Figure 12 shows how an *abstract network*, i.e. a collaboration diagram of abstract machines, can be defined. OBH is the object handling function.

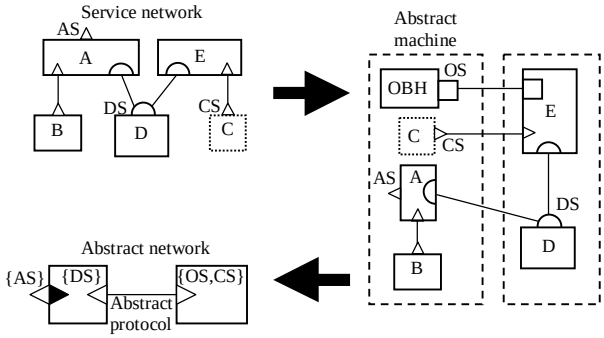

Figure 12. Abstract networks integrate service networks

Based on the service resource model in an interface, a set of *operations* are specified. All operations in an interface are performed by the provider. However, to perform a particular operation might require several interactions (i.e. ASPs) between the parties, e.g. one ASP where the user request the operation, and two alternative ASPs where the provider reports either the success or the failure of performing the operation.

Activities can be triggered by both parties, and they can be confirmed or not. By combining these characteristics, operations can be classified in six different *operation classes*, where a class defines if the outcome of the activity is confirmed or not, and which of the parties take the final decision on starting the activity. Figure 13 shows two of the six operation classes in AML. The idea of defining operation classes originates in the OSI remote operation standard, see (ITU-T 1988).

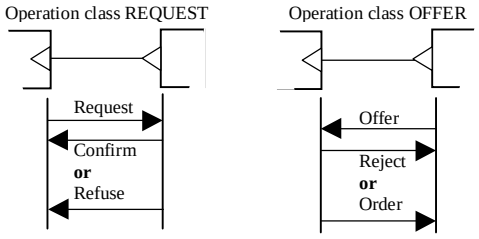

Figure 13. Operation classification

The OSI approach, and most object-oriented methods have a week support (if any) for describing interfaces with respect to operations that are initiated by the providing party. In many PCSs, however, this is actually the most common type of interaction, in special in interfaces of the PCS control structure. Typically (see Figure 13), a resource controller might detect an error situation in the resource system, and ask the user (with an *Offer*-ASP) if the controller shall execute a predefine operation to eliminate the problem. The user either rejects or accepts by responding with a *Reject*- or an *Order*-ASP. This is an operation of class *OFFER*.

Operation classes introduce a high semantic level in control structures. Each class and ASP define important control aspects, which are visible in behaviour descriptions, and which can be checked in run-time by standardized functions. All operation specifications for a particular primitive service are collected in the *primitive service specification.*

This specification also includes the service resource model and a *service behaviour* description. Resource models of the process control structure have a fundamental role also in identifying states of interfaces and service objects, and for describing behaviour. Each possible value of a service resource model is a potential *service state*, and each possible value of an object resource model is a potential *object state*. The way object resource models are related to service resource models (see Figure 5) guaranties that the behaviour of service objects can always be related to the behaviour in interfaces, and the other way around.

An elaborated state concept and state notation has been developed for AML. AML offers a notation for state diagrams where these state concepts are used. In order to avoid that a state diagram describes the implementation of an object (common in other state machine notations) instead of its behaviour in interfaces, the only symbols that are allowed in state diagrams are for states and *transitions*. Behaviour can also be expressed i terms of textual specifications of transitions (called *event specifications)*, that can be interpreted by the *AML-state machine*.

Sysnet modelling is supported by a method for translating abstract protocols to message protocols, by special notations for protocol networks, logical networks and physical networks, and by generic

solutions (patterns) for integration functions (object-, message- and association handling).

CONCLUSIONS

This paper has presented a model that can act as a unifying framework for modelling processing systems, i.e., systems that consist of a resource system and a process control system. The model was presented in two major views: A functional view and a solution view.

It was shown how the model of a resource system affects the functional structure of the process control system, and models of entities in that structure. It was also shown how the model can be applied on the telecom system domain.

The solution view considers all other functional requirements that are an effect of that the processing system might be distributed, and that it needs processing platforms in every physical node. The need for separating concerns is taken care of by introducing two architectural principles: Functional space and integration hierarchy. By these principles, the solution model can be separated into five model types, called service-, abstract-, protocol-, logical- and physical networks. The paper has described these architectural principles, and indicated how the model types are related.

A brief description of the modelling language AML was given. The basis for the development of AML has been the processing system paradigm that was presented in the paper.

Some relations between AML and UML are obvious. E.g., class diagrams can be used for resource system modelling in AML. In general, however, UML being a software system technology, does not support architectural modelling of sysnets, where component implementation (hardware or software) is indistinguishable, and where components communicate according to protocols and are organized in layered structures. Neither can abstract objects ("service objects" in AML) with their user-provider relations, operation classes and substructures be described in UML. It is subject to further research to identify possible adaptions of the UML, e.g. via its extension mechanisms (Hertzberg and von Wedel 1999).

AML has been verified by testing the language on the telecom systems domain for some years. Very promising results have been obtained in e.g. reverse engineering projects. For example, by using the protocol- and abstract network model types as tools for analysing a software implementation of a telecom platform, the software system was abstracted to an executable service network. Experiences from modelling network and system architectures, as well as network functions, are also very promising. Such models serve their purpose of being tools for system analysis; they can serve as implementation specifications and as tools for training system engineers.

A shallow investigation of AML may give the impression that AML is a complex language. However, AML is not complex, the systems we model are, which is also the reason for that modelling tasks become difficult, no matter how simple the language.

The only real problems we have faced with AML is the fact that the language is not known and standardized, and that AML-specific tool support is still missing. To publish AML is therefore an important step (Muth 2001). Although AML can be used as a conceptual tool only, to be able to exploit all potentials of AML, and to facilitate its use, tool support is needed. Efforts in that direction are in progress.

REFERENCES

Booch, Grady, “Software Engineering with Ada”. Benjamin/Cumming Publishing Co., Menlo Park (Calif.), 1983.

Fisher, J., “Model-Based Systems Engineering: A New Paradigm”. *INCOSE Insight,Vol 1,Issue 3, 1998.*

Herzberg, D. And Marburger, A., "An Extended Model for Real-Time systems". *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics (SCI) 2000*, pp 200-205.

Herzberg, D. and von Wedel, L., "*Erweiterungsmechanismen der UML. OJEKTspectrum Juli/August 1999*", pp 56-59.

ITU-T, “X.219, Remote Operations: Model Notation and Service Definition”. ITU-T, 1988.

Muth, T., "Modeling Telecom Networks and Systems Architecture. Conceptual Tools and Formal Methods." Springer Verlag Berlin Heidelberg 2001.

OMG, “UML 1.4”. OMG 2001. `www.omg.org`

Orfali, R., Harkey, D. and Edwards, J., “The Essential Distributed Objects Survival Guide”. John Wiley and Sons, Inc, 1996.

BIOGRAPHY

Thomas Muth holds a degree in electrical and electronic engineering. He joined Ericsson in 1970. He is the inventor of AML and the author of (Muth 2001). The last three years he has been working with Ericsson in Canada as a Senior Specialist in systems modelling. He now runs his own company.

Dominikus Herzberg holds a degree in engineering and a degree in economics. He joined Ericsson in 1995 and is working as a Senior Systems Designer.

Jens Larsen holds a degree in electrical and electronic engineering. He is the founding chairman of Software Egineering Process Special Interest Group at CRIM in Montreol, and a member of IEEE for over 22 years. He joined Ericsson in 1996 as a Senior Engineer and is working in the systems research department.