Mathematical System Models as a Basis of Software Engineering[⋆]

Manfred Broy

Institut für Informatik, Technische Universität München,
Arcisstr. 21, D–80290 München, Germany

Abstract. We give mathematical system models as a basis for system specification, system development by refinement, and system implementation. It provides a simple homogeneous mathematical and logical foundation of software and systems engineering. We treat mathematical concepts of refinement through levels of abstraction and complementing system views as they are used in software engineering. The goal is to give a coherent and simple mathematical basis.

1 Introduction

Software engineering comprises methods, description techniques and development processes for the development of large software systems. The full framework of a software engineering method (such as for instance SSADM, see [Downs et al. 92], or Cleanroom Software Engineering, see [Mills et al. 87]) contains a large amount of complex and highly interconnected information. Traditionally, this information is provided by so called reference manuals and rationals providing an often strange mixture of ideology, technical explanation, experience reports and ad hoc hints. Mostly, the meaning of the proposed description techniques remains partially unclear and so does their relationships.

We claim that it is possible to give a much more precise presentation of software engineering techniques. We show that this can be done without too much overhead by providing a mathematical basis in little more than ten pages.

Our general goal is to give a comprehensive mathematical foundation of the models and notions used and needed in software engineering (see [Booch 91], [Coad, Yourdan 91], [DeMarco 79], [Denert 91]) but keeping the mathematics as simple as possible. We describe a compositional system model that covers the main modeling issues dealt with in systems and software engineering.

1.1 Informal Survey of System Modeling Notions

An *interactive system* interacts with its environment by exchanging *messages*. The messages are exchanged through *input* and *output channels*. The causal

[⋆] This work was carried out within the Project SysLab, supported by Siemens Nixdorf and by the Deutsche Forschungsgemeinschaft under the Leibniz program. It is based on results worked out in the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitektur".

relationship between the input and output messages determines the *black box behavior* of a system also called its *interface*. Formally, this behavior is described by a *black box specification* also called an *interface specification*. By such a specification the behavior of a system may be specified uniquely for every pattern of input behavior given by its environment or the specification may leave some freedom. In the latter case we speak of *underspecification* or in the case of an operational system also of *nondeterminism*.

The behavior of a system may depend on the *timing* of its input messages. Also the timing of the output messages may be an important property of a system. Therefore we are interested in a specification technique that allows us to specify systems with timed and time dependent behaviors. For the description of the black box view of a system we use a logic based specification language.

When constructing an *implementation* of a system, we are not only interested in its black box behavior, but also in its internal structure. We speak of a *glass box view* of a system. Under its glass box view, a system may either be a *state machine* with a central state which we understand, in general, as a centralized nondistributed unit[1] or it may be a *distributed system* consisting of a family of *subsystems* called *components*. In a distributed system the only way the components interact is again by exchanging messages. However, also for a distributed system, a state view is possible by including all states of its components. This leads to a *distributed state*.

The messages and states of a system are mathematical elements of appropriately chosen carrier sets. They might be described by axiomatic specification techniques or by classical description techniques for *data models* as proposed in software engineering such as the widely used *entity/relationship* techniques.

A system (especially a distributed system) carries out a *process* that may depend on the behavior of the environment. Such a process consists of all the *actions* carried out by the system. Elementary actions of an interactive system consist in sending and receiving messages. The description of representative instances of such processes may help to understand the interactions of a system.

In the development of a system, we describe it and its parts at several *levels of abstraction*. Through the development, seen as a pure top down approach, we take into account more and more specific details and change the models such that they come closer to the structure required by system implementations finally leading to a *software architecture*. This process of system development is also called *refinement*. The notion of refinement is formalized by a refinement relation which is a mathematical relation between system specifications.

We consider, among others, the following types of refinement relations for system development:

- *black box refinement* (also called *property refinement*),
- *interface refinement*,
- *glass box refinement*.

[1] In this oversimplified view we include shared state systems with parallelism as nondistributed systems.

Glass box refinement aims at the design and implementation phase of a system. It may be classified into:

– *state space refinement*,
– *refinement by distribution*.

The corresponding refinement relations form the mathematical basis for the generation of logical verification conditions that have to be proved to show the correctness of the respective refinement steps.

1.2 Overall Organization of the Paper

In the following we define mathematical models capturing all the notions introduced informally in the introduction. We start by defining a mathematical system model which allows to model distributed systems in a hierarchical manner. Then we treat the notion of refinement and of complementing system views.

In our descriptions of system views and concepts, one goal is uniformity. We describe every system concept by a syntactic and a semantic part. In the syntactic part we define families of identifiers with additional sort information about them. In the semantic part, we associate mathematical elements with the introduced name spaces.

Our work is based on [Broy 91], [Focus 92], [Broy 93], [Broy 95] and [Rumpe et al. 95]. An application of mathematical models to a specific software engineering method is shown in [Hußmann 94] (see also [Hußmann 95]) by treating the British development method SSADM.

2 The Mathematical System Model

In this section we introduce a mathematical model for interactive and distributed systems and define a number of fundamental aspects and views.

2.1 Data Models

A *data model* is used to model the data occurring in an information processing system. It consists of a syntactic and a semantic part. The syntactic part consists of a *signature* $\Sigma = (S, F)$. S denotes a set of *sorts* and F denotes a set of *function symbols*. For each of these function symbols, a functionality is predefined by a mapping

$$\mathrm{fct} : F \to S^+$$

that associates with every function symbol its sequence of domain and range sorts. Thus, the syntactic part provides a name space with sort information.

Given a signature $\Sigma = (S, F)$, a Σ-algebra A consists of a carrier set s^A for every sort $s \in S$ and of a function

$$f^A : s_1^A \times \ldots \times s_n^A \to s_{n+1}^A$$

for every function symbol $f \in F$ with $\text{fct}(f) = <s_1 \dots s_{n+1}>$. A *sorted set* of identifiers is a set of identifiers X with a function

$$\text{Sort} : X \to S$$

that associates a sort with every identifier in X. By X^A we denote the set of all valuations which are mappings v that associate an element $v(x) \in \text{Sort}(x)^A$ with every identifier $x \in X$.

An *entity/relationship model* consists of a syntactic and a semantic part. Its syntactic part consists of a pair (E, R) where E is a sorted set of identifiers called *entities* and R is a set of identifiers called *relationships* for which there exists a function

$$\text{Sort} : R \to E \times E$$

The pair (E, R) is also called entity/relationship data model. A semantic model B of an entity/relationship model assigns a set e^B to each entity identifier $e \in E$ for which we have

$$e^B \subseteq \text{sort}(e)^A$$

and r^B is a *relation*

$$r^B \subseteq e^B \times e^B$$

A semantic model B is also called an *instance* of an entity/relationship data model. It is a straightforward step to include attributes into our concept of entity/relationship techniques. The set of all instances of an entity relationship model is called the *entity/relationship state space* (see [Hettler 94] for an intensive treatment of this subject). Note that this definition already includes a simple integrity constraint namely that every element occurring in a relation is also an element of the involved entity. Note, moreover, how easy it is in this formalization to combine entity/relationship models with axiomatic specification techniques (see [Wirsing 90]).

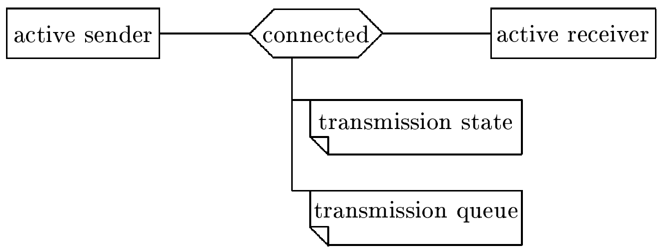

Fig. 1. Entity/relation diagram for the transmission medium with the entities *active sender* and *active receiver*, the relationship *connected* and two attributes

In Fig. 1 we show a simple entity/relationship diagram defining a data model for a transmission medium. It is a part of a message switching system which will be used as an example throughout the paper.

2.2 Communication Histories

Systems cooperate and interact by exchanging messages over channels. Given a sort of messages M, by

$$\text{Str } M$$

we denote the sort of *timed streams*. A timed stream is represented by a mapping

$$s : \mathbb{N}\backslash\{0\} \to (M^A)^\star$$

A stream denotes a *communication history* of a channel. We work with a discrete model of time and assume that our time is devided into an infinite sequence of time intervals. $s(i)$ represents the sequence of messages communicated in the ith time interval. Given a stream s, by

$$s|_k$$

we denote the restriction of the timed stream s to the first k time interval represented by $[1 : k]$. For every sequence of messages $m \in (M^A)^\star$ and every timed stream s of sort Str M we denote by

$$<m>\frown s$$

the stream with the sequence m as its first element (the sequence of messages communicated in the first time interval) followed by the stream s.

Given a sorted set of identifiers X for channels, a communication history for these channels is denoted by a function Val that associates a stream Val(c) of sort Str Sort(c) with every channel $c \in X$. The set of these communication histories for the sorted set of identifiers X is denoted by

$$\vec{X}$$

B $X^{A\star}$ we denote the set of mappings m that associate a sequence $m(c) \in (s^A)^\star$ with every channel $c \in X$ of sort $s = \text{Sort}(c)$. For every $m \in X^{A\star}$ that assigns a sequence of messages to every channel and every $x \in \vec{X}$ we denote by $<m>\frown x$ the communication history for the channels in X with

$$(<m>\frown x)(c) = <m(c)>\frown x(c)$$

for every channel $c \in X$.

2.3 Black Box System Models

A *black box system model* is given by a syntactic and a corresponding semantic interface. The syntactic interface consists of two sets of sorted identifiers I and O, denoting the sets of input and output channels with fixed sorts of messages communicated through them.

A black box behavior of a component with the syntactic interface (I, O) is modeled by a function

$$f : \vec{I} \to P(\vec{O})$$

(by $P(M)$ we denote the powerset over the set M) such that the output at time point k depends only on the input received till time point k. This is expressed by the following axiom of *well-timedness* (for all $i, j \in \vec{I}$):

$$i|_k = j|_k \Rightarrow f(i)|_k = f(j)|_k$$

A behavior f is called *deterministic*, if $f(i)$ contains exactly one element for every input history i. It is called consistent, if it contains a deterministic behavior.

The set of all black box behaviors with input channels I and output channels O is denoted by

$$I \rhd O.$$

Note that the set $I \rhd O$ provides the syntactic interface of a system and every element in $I \rhd O$ provides a semantic interface.

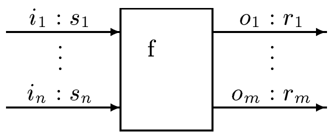

Fig. 2. Graphical representation of a syntactic interface with input channels $i_1, \ldots, i_n$ and output channels $o_1, \ldots, o_m$ and their respective sorts $s_1, \ldots, s_n$ and $r_1, \ldots, r_n$

2.4 State Transition Models

A state transition model is given by a nondeterministic state machine M with input and output. It consists of

- a state sort $s \in S$,
- an input set of sorted identifiers I,
- an output set of sorted identifiers O,
- a transition relation $\delta : s^A \times I^{A\star} \to P(s^A \times O^{A\star})$,
- a set $\sigma_0 \subseteq s^A$ of initial states.

With every state transition model we associate for every state $\sigma \in s^A$ a behavior

$$f_\sigma^M \in I \rhd O$$

by the following equation (let $i \in I^{A\star}$, $x \in \vec{I}$):

$$f_\sigma^M(\langle i \rangle \frown x) = \{\langle o \rangle \frown z : z \in f_{\tilde{\sigma}}^M(x) \wedge (\tilde{\sigma}, o) \in \delta(\sigma, i)\}$$

This is a recursive definition of the function f^M, but since the recursion is guarded its mathematical treatment is straightforward.

Fig. 3 gives a graphical representation of a system state view as it is often used in software engineering methods.

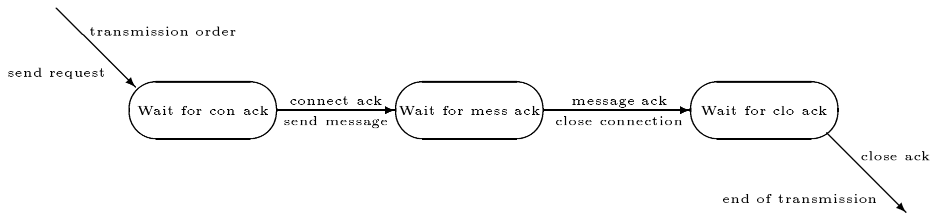

Fig. 3. State transition view of the sender, input messages are written above and output messages below the arrows

2.5 Distributed Systems

A distributed system $N = (C, I_0, O_0)$ consists of a set C of components that interact by exchanging messages over channels. Its syntax is given by

- the syntactic external interface of sorted input channels I_0 and output channels O_0,
- the set C of identifiers for components and a mapping that associates with each component identifier $c \in C$ a set of input channels I_c and a set of output channels O_c.

We require that all sets of output channels of the components in N are pairwise disjoint and disjoint to the set I_0 of input channels of the component and that $I_0 = H(N) \setminus \bigcup_{c \in C} O_c$. By $H(N)$ we denote the set of all channels of the system:

$$H(N) = I_0 \cup O_0 \cup \bigcup_{c \in C} (I_c \cup O_c)$$

The components and channels of a system form a data flow net. Fig. 4 gives an example of a graphical representation of a system by a data flow diagram. This provides a structural view of the system. For modeling dynamic systems where the number of components and channels changes over time, we need a more sophisticated mathematical model, of course.

The glass box semantics of the distributed system N is given by a mapping B that associates a behavior $B(c) \in I_c \rhd O_c$ with every component $c \in C$. A computation of the distributed system is a family of timed streams $x \in \overrightarrow{H(N)}$ such that

$$x|_{O_c} \in B(c)(x|_{I_c}) \text{ for all } c \in C$$

By $U(N)$ we denote the set of all computations of the distributed system N.

The black box behavior $B(N) \in I_0 \rhd O_0$ of the distributed system N with syntactic interface (I_0, O_0) is specified by (for all input histories $i \in \overrightarrow{I_0}$):

$$B(N)(i) = \{x|_{O_0} : x \in U(N) \wedge x|_{I_0} = i\}$$

$B(N)$ allows us to abstract away the distribution structure of the system N and to extract its black box behavior, its interface behavior.

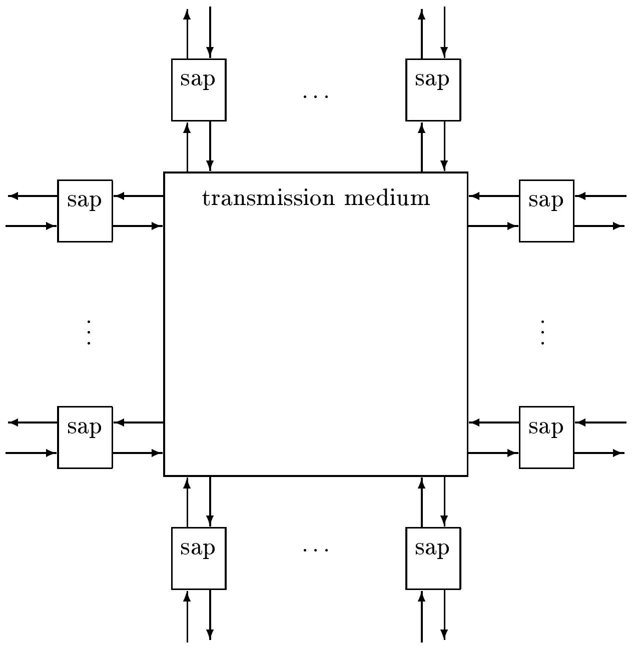

Fig. 4. A data flow diagram that gives a structural system view of a message transmission system with several service access points (saps)

2.6 Processes

For a distributed system N we have introduced its set of computations $U(N)$. In a computation we give for each channel a communication history by a stream. A process is a more detailed description of the run of a system N.

A process for an input i can be represented as a special case of a distributed system $\tilde{N} = (\tilde{C}, \tilde{I}, \tilde{O})$ with syntactic interface $(\tilde{I}, \tilde{O})$. For a process we assume an input $i \in \tilde{I}^A$ such that in the computation of $\tilde{N}$ for input i every channel (also those in $\tilde{I}$) contains exactly one message and the data flow graph associated with $\tilde{N}$ is acyclic and each component of P is deterministic. Then every channel denotes exactly one event of sending and receiving a message component $\tilde{c} \in \tilde{C}$ of the system $\tilde{N}$ denoting a process, represents one action. In this model of a process an action is a component that receives one input message on each of its input lines and produces one output message on each of its output channels. If there is a channel (an event) from an action a_1 to an action a_2 then a_1 is called *causal* for a_2.

This idea of modeling a process by a specific distributed system is similar to the concept of occurrence nets as introduced to represent concurrent processes that are runs of Petri-nets (see [Reisig 86]).

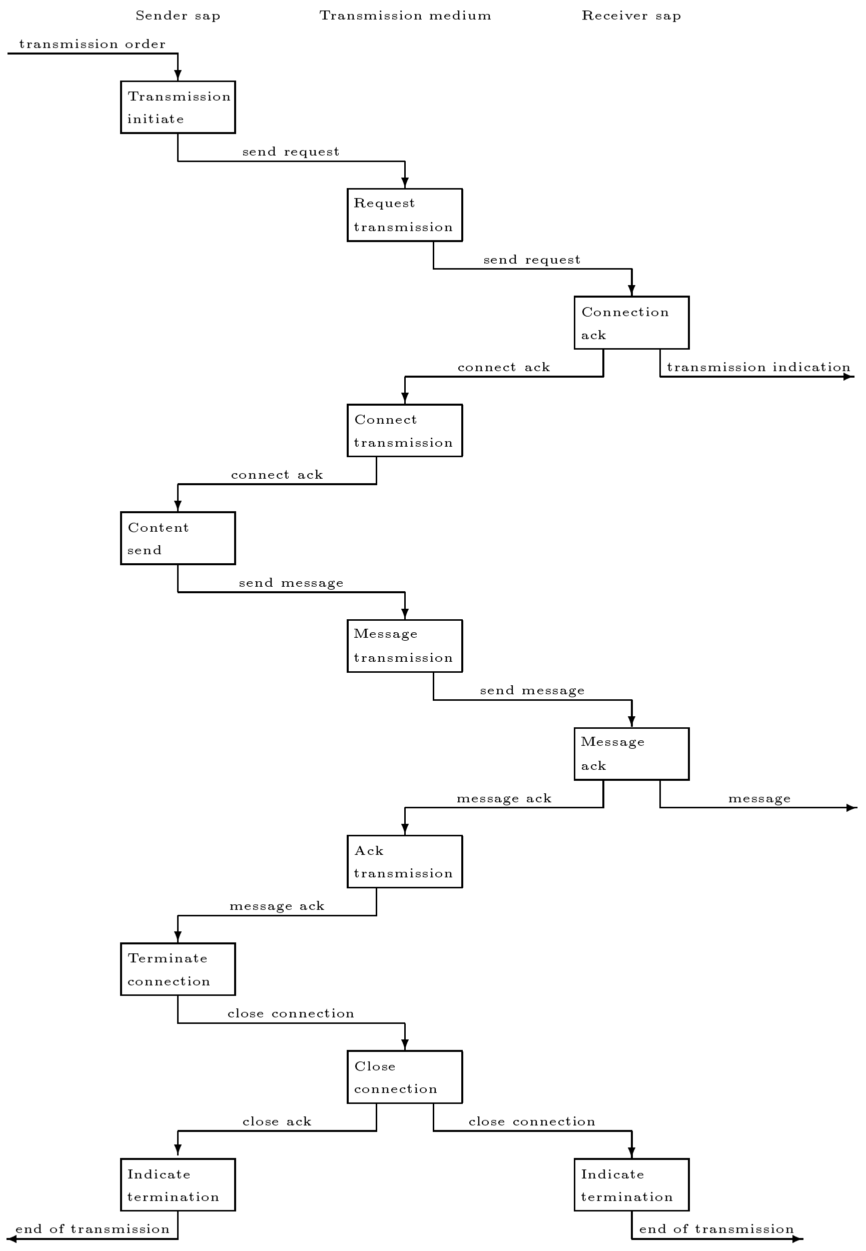

Fig. 5. Process description of a transmission scenario

2.7 Complete System Models

A complete hierarchical system model is given by a black box view consisting of a syntactic and semantic interface and of a glass box view consisting of either a corresponding state transition system or of a corresponding distributed system for it. In the latter case we require that each of its component is a complete hierarchical system again. This yields hierarchical distributed systems. Since we can associate a black box behavior to every state transition system and every distributed system, we can associate a behavior to every component in the hierarchy if the nesting is finite. A complete system is called *interface consistent*, if for each component its glass box behavior is consistent with (or a refinement of, see below) its given semantic interface.

3 Refinement

Large complex systems cannot be developed by considering all its complex properties in one step. Following the principle, not to cover more than one difficulty at a time, refinement allows us to add complexity to system models stepwise in a controlled way. All approaches to software engineering work with a formal or an informal concept of refinement.

We formalize the idea of refinement with the help of refinement relations in the sequel. A refinement relation is formally a mathematical relation between mathematical system models.

3.1 Refinement of Data Models

A data model given by a Σ-algebra A can be refined by adding sorts and function symbols to the signature and respectively carrier sets and functions. A more general notion of *refinement* is obtained by renaming the signature $\Sigma = (S, F)$ into a signature $\tilde{\Sigma} = (\tilde{S}, \tilde{F})$ by a *signature morphism*. It is given by a pair of functions

$$\sigma_1 : S \to \tilde{S}, \sigma_2 : F \to \tilde{F}$$

where $\tilde{\Sigma} = (\tilde{S}, \tilde{F})$ is the refined signature and for all $f \in F$:

$$\text{fct } \tilde{f} = \sigma_1(\text{fct } f)$$

where the mapping σ_1 is extended to sequences of sorts elementwise. This determines the syntactic part of refinement. A $\tilde{\Sigma}$-algebra $\tilde{A}$ is called a *refinement* of the Σ-algebra A, if there are functions

$$\alpha_s : \sigma_1(s)^{\tilde{A}} \to s^A, \varrho_s : s^A \to P(\sigma_1(s)^{\tilde{A}})$$

for every sort $s \in S$ such that for every data element $a \in s^A$ we have:

$$\{\alpha_s(\tilde{a}) : \tilde{a} \in \varrho_s(a)\} = \{a\}$$

and for all functions $f \in F$ with $\text{fct}(f) = <s_1 \dots s_{n+1}>$ we have for all data elements $a_1 \in s_1^A, \dots, a_n \in s_n^A$:

$$\alpha_{s_{n+1}}(\sigma_2(f)^{\tilde{A}}(\tilde{a}_1, \dots, \tilde{a}_n)) = f^A(a_1, \dots, a_n)$$

for all $\tilde{a}_1 \in \varrho_{s_1}(a_1), \dots, \tilde{a}_n \in \varrho_{s_n}(a_n)$. This is the classical notion of data refinement, where all abstract elements are represented by concrete elements. This way we obtain a refinement notion for state machines and also for entity/relationship models (for a detailed treatment of this aspect see [Hettler 94]).

3.2 Refinement of Communication Histories

The refinement concept for general data models can be carried over to a refinement concept for communication histories. This is an advantage of the incorporation of communication histories as mathematical elements into our system model. Given a pair of functions

$$\alpha : \overrightarrow{X}_1 \to \overrightarrow{X}_0, \varrho : \overrightarrow{X}_0 \to P(\overrightarrow{X}_1)$$

a sorted set C_1 of identifiers is called a *communication history refinement* of the sorted set of identifiers X_0 if we have

$$\{\alpha(\tilde{c}) : \tilde{c} \in \varrho(c)\} = \{c\}$$

for all $c \in \overrightarrow{C}_0$. Since we use an explicit representation of communication histories in our system models, the refinement notion is a simple generalization of our refinement notion for data models.

3.3 Refinement of Black Box Views

A refinement relation for black box system models is defined by a relation between systems. Given two component behaviors $f_0, f_1 \in I \rhd O$ with the syntactic interface (I, O) the behavior f_1 is called a *black box refinement* of f_0 if for all input histories $i \in \overrightarrow{I}$ we have

$$f_1(i) \subseteq f_0(i)$$

We generalize this simple notion of refinement to *interface refinement* as follows. Assume the functions

$$\alpha_1 : \overrightarrow{I}_1 \to \overrightarrow{I}_0, \varrho_1 : \overrightarrow{I}_0 \to P(\overrightarrow{I}_1)$$

$$\alpha_2 : \overrightarrow{O}_1 \to \overrightarrow{O}_0, \varrho_2 : \overrightarrow{O}_0 \to P(\overrightarrow{O}_1)$$

that define a communication history refinement I_1 for I_0 and a communication history refinement O_1 for O_0, then the black box behavior

$$f_1 \in I_1 \rhd O_1$$

is called an *interface refinement* of the black box behavior

$$f_0 \in I_0 \rhd O_0$$

if for all input histories $x \in \vec{I}_0$

$$\{\alpha_2(f_1(x)) : x \in \varrho_1(x)\} = f_0(x)$$

Again this is just a straightforward generalization of the concept of data model refinement to the behavior of systems. It allows us to refine systems to systems with a different number of input and output channels, different names and with different sorts that may lead to a different granularity of messages. A simple example is the refinement of a system working with numbers (for instance an adder) into a system working with bits (for more details, see [Broy 93]).

3.4 Refinement by Distribution

A distributed system N with interface (I, O) is called a *refinement by distribution* of a black box behavior $f \in I \rhd O$ if $B(N)$ is a refinement of f. $B(N)$ denotes the black box behavior of a system that is defined as described in section 2.5.

3.5 Process Refinement

A process is represented by a special case of a distributed system. So all refinement notions introduced for distributed systems carry over to processes. Hence a process p is a refinement of an action a, if p is a refinement by distribution of the action a. Recall that an action is a special case of a system.

3.6 Glass Box Refinement

Given a distributed system N with a specified black box behavior for all its components, a glass box refinement associates a state machine or a distributed system with a behavior that is a refinement of the specified one. Hierarchical iterated glass box refinement leads to a complete system model.

4 System Views

For the development of large complex systems it is helpful to work with complementary system views. A system view is a projection of a system onto a particular aspect. For a given distributed system we find it useful to work with the following views:

- process views,
- data model and state views,
- black box views (interface views),
- structural views.

Views allow us to concentrate on specific aspects of a system: Given a complete distributed system we define in the following the mentioned views for it.

4.1 Data Model View

For a complete distributed system $N = (C, I, O)$ with syntactic interface (I, O) a data model view provides for each of its components $c \in C$ a data view. Then the state space consists of an assignment that associates a state sort with each component in C. The corresponding state sort can be used to define a state transition system, or, if the component is again a distributed system, a distributed data view can be defined for it.

4.2 Black Box View

Both for state transition systems and distributed systems we have specified a black box view by associating a behavior with them. This provides black box views for all kinds of glass box views of systems that we have considered so far.

4.3 Structural Views

A structural view onto a distributed system N is given by its set of components and the channels connecting them. The structural view allows us to draw a data flow diagram showing the structuring of a system into its subsystems (its components) and their communication connections (channels).

4.4 Process Views

For a distributed system $N = (C, I, O)$ with syntactic interface (I, O) a process view for an input $i \in \vec{I}$ is given by process represented by a distributed system $\tilde{N} = (\tilde{C}, \tilde{I}, \tilde{O})$ consisting of a set $\tilde{C}$ of actions (components). The relation between the distributed system N and the process $\tilde{N}$ is given by two functions

$$\text{act} : \tilde{C} \to C, \text{chan} : H(\tilde{N}) \to H(N)$$

We assume that for each channel $c \in H(N)$ the set of process channels (representing events)

$$\{\tilde{c} \in H(\tilde{N}) : \text{chan}(\tilde{c}) = c\}$$

associated with it is linearly ordered. We assume that for all channels $\tilde{c} \in H(N)$ we have

$$\text{chan}(\tilde{c}) \in I \iff \tilde{c} \in \tilde{I}$$

$$\text{chan}(\tilde{c}) \in O \iff \tilde{c} \in \tilde{O}$$

Further more we assume that $\text{chan}(\tilde{c})$ is in the input or output channels of a component $\tilde{c} \in \tilde{C}$ if $\text{chan}(c)$ is in the input or output channels respectively of the component $\text{act}(\tilde{c})$.

The process $\tilde{N}$ is called a process view for system N with input i if there exists a computation x of N for the input history i such that for every computation $\tilde{x}$ of $\tilde{N}$ the streams associated with x carry the messages occurring in the linear order for the channels of $\tilde{N}$ that are mapped on the respective channels.

5 Conclusion

We have provided a family of mathematical models and concepts that can be used as the core of a mathematical basis for software engineering. Methodological and descriptional concepts of a method can be precisely defined in terms of these models. It is our goal to demonstrate, how simple and straightforward such a mathematical model is. It shows, in particular, that software engineering methods can be provided with a tractable mathematical basis without too much technical overhead.

There are many specific areas where mathematical system modeling can be useful to give more precision to software engineering areas. Examples are software architectures (see [Garlan, Shaw 93]), formal methods for the development of large software systems (see [Abrial 92]) or systematic program development methods (such as [Jones 86]). The structures introduced above can be used, in particular, for the Cleanroom Software Engineering approach propagated in [Mills et al. 87].

References

[Abrial 92]	J.R. Abrial: On Constructing Large Software Systems. In: J. van Leeuwen (ed.): Algorithms, Software, Architecture, Information Processing 92, Vol. I, 103-119
[Booch 91]	G. Booch: Object Oriented Design with Applications. Benjamin Cummings, Redwood City, CA, 1991
[Broy 91]	M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL. Formal Aspects of Computing 3, 21-57 (1991)
[Broy 93]	M. Broy: (Inter-)Action Refinement: The Easy Way. In: M. Broy (ed.): Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, pp. 121-158, Berlin, Heidelberg, New York: Springer 1993
[Broy 95]	M. Broy: Advanced Component Interface Specification. In: Takayasu Ito, Akinori Yonezawa (eds.). Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings, Lecture Notes in Computer Science 907, Springer 1995
[Coad, Yourdan 91]	P. Coad, E. Yourdon: Object-oriented Analysis. Prentice Hall International Editions 1991
[DeMarco 79]	T. DeMarco: Structured Analysis and System Specification. Yourdan Press, New York, NY, 1979
[Denert 91]	E. Denert: Software-Engineering. Springer 1991
[Downs et al. 92]	E. Downs, P. Clare, I. Coe: Structured analysis and system specifications. Prentice Hall 1992
[Focus 92]	M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, R. Weber: The Design of Distributed Systems - an Introduction to Focus. Technical University Munich, Institute of Computer Science, TUM-I9203, Januar 1992, see also: Summary of Case

Studies in Focus - a Design Method for Distributed Systems. Technical University Munich, Institute for Computer Science, TUM-I9203, Januar 1992

[Garlan, Shaw 93] D. Garlan, M. Shaw: An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering. 1993

[Hettler 94] R. Hettler: Zur Übersetzung von E/R-Schemata nach Spectrum. Technischer Bericht TUM-I9409, TU München, 1994

[Hußmann 94] H. Hußmann: Formal foundation of pragmatic software engineering methods. In: B. Wolfinger (ed.): Innovationen bei Rechen- und Kommunikationssystemen, Informatik aktuell, Berlin: Springer, 1994, 27-34

[Hußmann 95] H. Hußmann: Formal Foundations for SSADM. Technische Universität München, Fakultät für Informatik, Habilitationsschrift 1995

[Jones 86] C.B. Jones: Systematic Program Development Using VDM. Prentice Hall 1986

[Mills et al. 87] H. Mills, M. Dyer, R. Linger: Cleanroom Software Engineering. IEEE Software Engineering, 4:19–24, 1987

[Reisig 86] W. Reisig: Petrinetze - Eine Einführung. Studienreihe Informatik; 2. überarbeitete Auflage (1986).

[Rumpe et al. 95] B. Rumpe, C. Klein, M. Broy: Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technische Universität München, Institut für Informatik, 1995, TUM-I9510

[Wirsing 90] M. Wirsing: Algebraic Specification. In: J. van Leewwen (ed.): Handbook of Theorectical Computer Science, Volume B, chapter 13, pages 675–788, North-Holland, Amsterdam 1990

Biographical Paragraph

Prof. Dr. Manfred Broy
Fakultät für Informatik
Technische Universität München
D–80290 München

Prof. Dr. Manfred Broy is full professor of computing science at the Technical University of Munich. His research interests are software and systems engineering comprising both theoretical and practical aspects. This includes system models, the specification and refinement of system components, specification techniques, development methods, advanced implementation languages, objectorientation, and quality assurance by verification. He is leading a research group working in a number of industrial projects that try to apply mathematically based techniques and to combine practical approaches to software engineering with mathematical rigour.

Professor Broy is the organizer of the Marktoberdorf Summer Schools in foundations of programming. He published a four volume introductory course to computing science (in German). He is main editor of Acta Informatica and editor of Information and Software Technology, Distributed Computing, Formal Aspects in Computer Sciences, and Journal of Universal Computer Science.

Professor Broy is a member of the European Academy of Science. In 1994 he received the Leibniz Award by the Deutsche Forschungsgemeinschaft.

This article was processed using the LaTeX macro package with LLNCS style