Formal description techniques - how formal and descriptive are they?*

M. Broy
Institut für Informatik
Technische Universität München
80290 München, Germany

Abstract

I discuss formal description techniques (FDTs) as they are applied in practice in software and system engineering. Their quality can be measured by their formality, descriptiveness and technical usefulness. I discuss shortcomings in the scientific and semantic foundations of FDTs. I formulate requirements for FDTs. I demonstrate how a family of complementary description techniques can be developed on the basis of a mathematical system model that provides a scientific methodological foundation for the modular specification and development of systems. This demonstrates a proceeding in the design of FDTs that helps to make them formal, descriptive and technically useful.

Keyword

Formal description techniques, mathematical semantics, mathematical system model

1 INTRODUCTION

The descriptive and formal specification of the behaviour of reactive information processing systems is of high relevance in many technical applications. Typical application areas are telecommunication and embedded reactive systems as they are used for instance in automotive or avionics applications. For these systems often a complex behaviour is required which includes a close co-operation between the system and its environment. In general, hard or at least soft real time aspects are involved.

As it is generally agreed, specifying such systems cannot be done by a verbal description runs into major difficulties. Verbal descriptions tend to be lengthy, incomplete, to contain phrases that may be misinterpreted, and to be not well structured. Moreover, they do not follow any description standards. Therefore formal description techniques (FDTs) have been suggested as a better way to describe complex reactive systems with respect to their information processing behaviour. Typical examples for such FDTs are, among many others,

* This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the BMBF-project KorSys and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program

SDL, Esterel, statecharts, Lotos, Lustre, and Estelle. They provide a graphical and/or textual syntax for the description of reactive systems.

However, typically for FDTs only their syntax is formally defined but not their exact meaning. For most of the FDTs used in practice there exist semantic shortcomings (see, for instance, Beeck (1994) on the semantics of statecharts or Hinkel (1995) on the time model of SDL).

2 REQUIREMENTS FOR FDTs

FDTs are used for several purposes. Their main use is seen in providing a description of the behaviour of complex reactive systems. Such descriptions are needed in several phases of the development of a system. These phases comprise roughly

- requirements capture,
- design of the system architecture,
- implementation of the system components.

Depending on the development phase for which a FDT is used, different aspects have to be emphasised. In the requirement phase the flexible specification of systems in terms of their properties is most important. For describing the system architecture the emphasis lies on the description of the components of a system, their interface, and how the components interact. In the implementation phase the components have to be described in such a style that the required behaviour can be generated by machines with the help of compilers and interpreters. So in the requirement and design phase a FDT must be a specification language, while in the implementation phase it must be a programming language. This shows that the requirements for FDTs are quite different depending on their purpose of usage.

On the other hand, since both in the requirement phase and in the design phase it is often useful to generate rapidly prototypes, operational concepts are useful for these phases, too. It is often useful to combine purely property oriented with operational descriptions. In all phases readability, understandability, and expressiveness are urgently required. FDTs support the communication between human beings, for unambiguously documenting development results and for their analysis. So the ideal FDT combines property oriented and operational concepts. A flexible FDT gives us the freedom to leave certain aspects deliberately unspecified by including the possibilities of underspecification; in operational terms this leads to nondeterminism.

I require the following principles for a FDT that make it a useful tool in system development:

- formal syntax,
- formal semantics,
- clear conceptual system model,
- uniform notion of an interface,
- sufficient expressiveness and descriptive power,
- concept of development techniques with a proper notion of refinement and implementation.

Unfortunately so far these requirements are not fulfilled for most of the formalisms used in practice. In many cases a formal semantics is not given at all and the informal semantics leaves a lot of questions open. In cases where a formal semantics is provided, it often does not coincide with the informal description, it often is not powerful enough to express certain issues such as fairness, or it is too complicated to be helpful. Finally, often the formal semantics is seen as a second order issue and not carefully approved by standardisation activities.

A method for giving a semantic basis for a FDT should be powerful enough to express its semantics in a style easy to read and to comprehend. It should guide the design of the FDT but not constrain it unnecessarily. A typical example are message sequence charts that have recently been standardised. To describe their meaning formally process algebras have been used. As a result, several ideas of the designers have been rejected on the basis that it would be very difficult to describe the meaning of those ideas by process algebras. This is certainly not what a semantic technique should do. It should provide guidance but it should not prevent the designer of description formalisms to drop well understood concepts only because the semantic framework is not powerful enough. What we need is a powerful semantic framework that is, nevertheless, easy to comprehend which gives additional guidance that is helpful in avoiding unsuggestive concepts and unsuggestive descriptions. Much more work is needed to provide such powerful models. In addition, we need to be able to provide a scientific basis for a careful analysis of the description concepts including timing concepts that are most appropriate for the various application areas of system engineering.

For embedded systems and telecommunication systems typically soft or even hard real time properties are part of the requirements. As the consequence the modelling and description techniques have to be powerful enough to express real-time aspects. This has two important consequences for formal description techniques:

- An adequate concept has to be found that allows to model the relevant behaviours such as reactions within time bounds, interrupts, pre-emptions, or delays.
- A semantic model has to be found that represents the behaviour in a time frame in a way that provides a precise description of what the timing in specifications really means.

Both aspects are crucial for description formalisms and weak points of most of the existing formalism. For instance, in statecharts there exists a confusing variety of different models for timing as outlined in Beeck (1994). For Esterel (see Berry, Gonthier (1988) and Broy (1996)) there is a long standing discussion on the semantics of Esterel especially how to model the zero-delay-assumption.

For SDL the timing issue is also not at all clear. From the reference manual one could get the impression that there is no restriction on the timing of signal transmission not even on the transmission of time signals. This implies that signals can be arbitrarily delayed and therefore no behaviour can be specified by a SDL description which is guaranteed to function within pregiven time bounds. In general, the modelling of time in SDL is not explained very clearly in the reference manual. Therefore it is easy to formulate descriptions in SDL where even the experts start to debate what the behaviour of the system described by SDL should be.

This leads to a clear conclusion:

- The semantics of description formalism have to be described by clear mathematical means that allow a unambiguous description of the meaning of the constructs including the time model.
- The basic semantic model has to be powerful enough to reflect all the relevant issues in the semantics including especially time.

The semantic model has to be clear and easy to comprehend such that is gives an effective basis for the understanding of the description formalism. It has to support concepts such as underspecification and nondeterminism which are valuable prerequisites for a development methodology.

In the following I intend to demonstrate how a FDT can be defined starting from semantic notions. This leads to clean semantic concept in contrast to FDTs that have been designed starting form syntactic notions and only later trying to provide semantic descriptions. I also show how development notions such as refinement can be based on this concept.

3 MATHEMATICAL MODELS OF SYSTEMS AND THEIR DESCRIPTION

A complex information processing system cannot be described in a monolithic way if understandability is an issue. It is better to structure its description into a number of complementing views. In this section I give a brief overview over the particular aspects of the type of systems I am dealing with. These include:

- data models,
- system component models (interface models),
- distributed system models,
- state transition models,
- process models.

For each of these aspects of a system a mathematical model is sketched in the following, consisting of a syntactic and a semantic part. I use these models as a basis for more pragmatic graphical system description formalisms.

The integration of these description formalisms and their mathematics is an indispensable requisite for more advanced system and software development methods. In the following I discuss some of the widely used description formalisms in software engineering and their mathematics. In the development process the various description formalisms serve mainly the following purposes:

- means for the analysis and requirement capture for the developer,
- basis for the validation of development results,
- communication between the engineers and the application experts,
- documentation of development results and input for further development steps.

Of course, the usefulness of a description formalism has always to be evaluated with respect to these goals. Software engineering has provided many different description techniques for the various aspects of a system.

A very universal and precise method for the description of properties and aspects of a system is mathematical logic. A logical formalism provides a formal syntax, a mathematical semantics and a calculus in terms of a set of deduction rules. The later can be used to derive further properties from a set of given properties. Proper description formalisms also have a formal syntax and a mathematical semantics. In contrast to logical formalisms the description formalisms of software engineering are not especially designed for the formal manipulation by deduction and transformation rules. Obviously, there is a close relationship between description formalisms and mathematical logic. Strictly speaking, a description formalism formulates a property of a system. So it can be understood as a predicate. Consequently, we may look for rules that allow us to translate description formalisms into logical formulas. This allows us to use pragmatic description techniques without having to give up the preciseness of mathematical logic and its possibilities of formal reasoning.

4 DATA MODELS AND THEIR SPECIFICATION

Data models are needed to represent the information and data structures involved in an application domain. Often data models do mainly capture the structure of the data and their relationship but not their characteristic operations. These, of course, should by an integral part of the data model. Therefore I understand a data model always as a family of data sets together with their relationships and characteristic functions.

Families of sets with operations on them are known in mathematics as algebra. From a mathematical point of view, a data model consists of a heterogeneous algebra given by a family of carrier sets and a family of functions. More technically, I assume a set of *sorts*[1] (often also called *types* or *modes*) S and a set F of constant symbols including function symbols with a fixed functionality

$$\mathbf{fct} : F \rightarrow S$$

The function **fct** associates with every function symbol in F its domain sorts and its range sort. Both the sets S and F provide only names. I assume that the set S contains besides basic sorts also tuples of sorts as well as functional sorts and even polymorphic sorts (such as Set α denoting the sort of finite sets of elements of sort α for each α in S, see SPECTRUM in Broy et al. (1993) for details). The pair (S, F) together with the function **fct** that assigns functionalities to the identifiers in F is often called the *signature* of the algebra. The signature is the static part of a data model and provides a syntactic view.

In every algebra A of signature (S, F) I associate with every sort $s \in S$ a carrier set s^A (a set of data elements) and with every function symbol $f \in F$ a constant or function f^A of the requested sort or functionality. It is typical that mathematical structures modelling information processing concepts include static (syntactic) parts such as name spaces (in the algebraic case the signature) and semantic parts (in the algebraic case sets and functions).

In the data view, I fix the relevant sorts and their characteristic functions for the system I want to describe or implement. In addition, I may describe the states of system components. This can be achieved by recursive sort declarations based on enumeration, tupling (records), and disjount union (variant types) as we find them in programming languages, by axiomatic data structure specifications, and/or by E/R-diagrams, especially when our system processes mass data. Often data models are used to describe the state of systems and components.

Data models can be described by logic using the *axiomatic* or of *algebraic* specification of data structures. The techniques of algebraic specification is in the meanwhile well-understood (see Wirsing (1990) for an overview or SPECTRUM in Broy et al. (1993) for an instance of a particular algebraic specification language).

Data models can especially be used to model the state of a component. Given a signature (S, F) as a basis I define a state space by a signature (S, F $\cup$ V). The symbols in V are called the *variables* or the *attributes* of a state. They have a sort as well as the symbols in F.

A typical technique to describe state spaces are entity/relationship-diagrams. I denote entity/relationship by symbols in diagrams as shown in Figure 1.

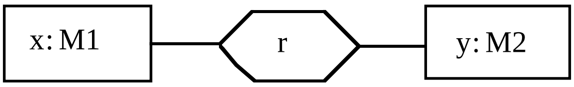

Figure 1 Two Entities Connected by One Relationship

[1] We believe, like many others computing scientists and software engineers, that data sorts (typing) is a very helpful concept in modeling application and software structures.

The diagram given in Figure 1 is used to express that x, r, y are state attributes with the sorts

x: Set M1
y: Set M2
r: Rel[M1, M2]

where Rel[M1, M2] is a polymorphic sort declared by:

sort Rel[α, β] = Set Pair(c1:α, c2:β)

Of course, I have to add invariants specifying that the attribute r contains only pairs (a, b) as elements such that a is in x and b is in y respectively.

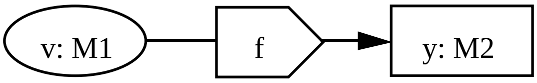

Figure 2 Functional Relationship

I write the diagram shown in Figure 2 to express that v is an attribute of the data space

v: M1

and f is a function attribute

f: M1 → M2

that maps v onto an element in the entity y.

FDTs with graphical representations and diagrams can be used as an illustrative, but nevertheless fully formal description technique, if both a formal syntax and a mathematical semantics are provided. A useful form to do this is a translation into axiomatic specifications. This allows us to integrate formal axiomatic data description techniques and pragmatic graphical description techniques into one homogenous software engineering method. A convincing translation of E/R-concepts into axiomatic specifications is given by Hettler (1994).

5 SYSTEM COMPONENTS

In this section I introduce a mathematical concept of a system component and techniques for describing such components. From a practical point of view, a component is considered as a black box that encapsulates related services according to a published specification.

I am interested in system models that allow us to represent systems in a modular way. I think of a system as consisting of a number of subsystems that I call *components*. Moreover, a system itself is a component again which can be part of a larger system. A component is a self-contained unit with a clearly specified interface. Through and only through its interface it is connected with its environment. In this section I introduce a simple, very abstract mathematical notion of a system component. Accordingly, a (system) *component* is an information processing unit that communicates with its environment through a set of input and output channels. This communication takes place in a (discrete) time frame.

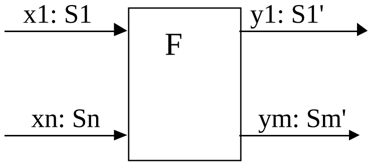

Figure 3 Graphical Representation of a Component as a Data Flow Node with input Channels x1, ... , xn and Output Channels y1, ... , ym and their Respective Sorts

In software engineering, it is helpful to work with a *black box view* and a *glass box view* of a component. In a black box view we only are interested in the interface of a component with its environment. In a glass box view we are interested in the internal structure of a component, be it its local state space or its decomposition into subcomponents. I first give a model for the black box view.

Let I be the set of input channels and O be the set of output channels. With every channel in the set $I \cup O$ we associate a data sort indicating the sort of messages sent on that channel. Then by (I, O) the *syntactic interface* of a system component is given. For simplicity, I use just one set M of data sorts for messages on the channels in the sequel in the theoretical parts of the presentation to keep the mathematics more readable. A graphical representation of a component with its syntactic interface is shown in Figure 3.

Let M be a sort of messages and signals. By M^* we denote the set of finite sequences over the set of messages M, by M^∞ we denote the infinite sequences over the set of messages M. By ˆ we denote the concatenation of sequences. The set M^∞ can be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. Formally I define the set of timed streams by (I write S^∞ for the function space $\mathbb{N}^+ \rightarrow S$ and $\mathbb{N}^+$ for $\mathbb{N}\backslash\{0\}$)

$$M^{\aleph} =_{\text{def}} (M^*)^\infty .$$

For every set of channels C, every mapping $x: C \rightarrow M^{\aleph}$ provides a complete communication history. Note that $(C \rightarrow M^*)^*$ and $C \rightarrow (M^*)^*$ are isomorphic. Moreover, the set $M^{\aleph}$ is isomorphic to the set of streams over the set $M \cup \{\surd\}$ which contain an infinite number of time ticks (here $\surd$ denotes a time tick; I assume $\surd \notin M$).

I denote the set of valuations of the channels in C by sequences

$$C \rightarrow M^* \qquad \text{by} \qquad \vec{C}^*$$

I denote the set of the valuations of the channels in C by infinite timed streams

$$C \rightarrow M^{\aleph} \qquad \text{by} \qquad \vec{C}$$

I denote for every number $i \in \mathbb{N}$ and every stream $x \in M^{\aleph}$ by

$$x\downarrow i \in (M^*)^*$$

the sequence of the first i sequences in the stream x. It represents the observation of the communication over the first i time intervals. By

$$\overline{x} \in M^* \cup M^\infty$$

I denote the finite or infinite stream that is the result of concatenating all the finite sequences in the stream x. $\overline{x}$ is a finite sequence if and only if a finite number of sequences in x are nonempty. Going from the stream x to $\overline{x}$ provides a time abstraction. In the stream x, it is represented in which time interval which messages arrive, while in $\overline{x}$ only the messages in their order of communication are described. Both notations $x\downarrow$ and $\overline{x}$ as well as the concatenation introduced for streams x can be lifted to tuples and sets of timed streams by applying them pointwise.

Figure 3 describes the syntactic interface of a component with the input channels $x_1, ..., x_n$ of the sorts $S_1, ..., S_n$ and the output channels $y_1, ..., y_m$ of sorts $S_1', ..., S_m'$.

I represent the behaviour of a component with the set of input channels I and the set of output channels O by a function:

$$F: \vec{I} \to \wp(\vec{O})$$

This function yields the set of output histories F.x for each input history x. To give a precise definition of the properties required for a function representing a component I introduce a number of notions. A function

$$F: \vec{I} \to \wp(\vec{O})$$

is called

- *timed*, if for all $i \in \mathbb{N}$ the following formula holds:

$$x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i = F(z){\downarrow}i$$

Then the output in the time interval i only depends on the input received till the i´th time interval.

- *time guarded*, if for all $i \in \mathbb{N}$ the following formula holds:

$$x{\downarrow}i = z{\downarrow}i \Rightarrow F(x){\downarrow}i{+}1 = F(z){\downarrow}i{+}1$$

Time guardedness assumes in addition to timedness that the reaction to input is always delayed by at least one time unit.

- *fully consistent*, if there exists a time guarded function[1] f: $\vec{I} \to \vec{O}$, such that for all input histories x:

  $$f.x \in F.x$$

  By $[\![F]\!]$ I denote the set of strongly time guarded functions f with $f.x \in F.x$ for all x.

- *fully consistent*, if for all input histories x:

$$F.x = \{f.x: f \in [\![F]\!]\}$$

I assume in the following that stream processing functions that represent the behaviour of components are always time guarded and fully consistent. I write for the set of functions

$$F: \vec{I} \to \wp(\vec{O})$$

that are time guarded and fully consistent

$$Ç[I, O]$$

Ç[I, O] denotes the set of all behaviours with the syntactic interface consisting of the input channels I and the output channels O. By Ç I denote the set of all component behaviours.

6 STATE TRANSITION DESCRIPTIONS

In this section I introduce a mathematical concept of a state transition system and techniques for describing it. State machines are a well-known system model. They are used both in practice and in theory. By a state machine some internal details of the system in terms of the states are provided. Therefore I say that state machines provide a *glass box view* of a system component. In the following section I provide another type of a glass box view by distributed systems.

A mathematical state based system model uses state transitions. It includes in addition to the sets of input and output channels that form the syntactic interface above a set State which

[1] Since functions can be seen as a special case of set valued functions where the result sets contain exactly one element, the notion of time guardedness extend to functions.

denotes the set of states (the state space) of the component. I define a state transition machine by a state transition function

$$\Delta: \text{State} \times \vec{I}^* \to (\text{State} \times \vec{O}^* \to \text{Bool})$$

and a set

$$\text{State}_0 \subseteq \text{State}$$

of initial states.

A state transition machine is nondeterministic, in general. In each transition step it takes a state and a communication pattern of its input streams and yields a predicate that characterizes the piars consisting of a successor state and a communication pattern for its output streams. A state machine models the behaviour of an information processing unit with input channels from the set I and output channels from the set O in a time frame as follows. Given a family of finite sequences $x \in \vec{I}^*$ representing for every channel $c \in I$ the sequence x(c) of input messages received in a time interval of the component in state $s \in$ State, every pair (s‘, y) in the set $\Delta(s, x)$ represents a possible successor state and the sequence of output messages y(c) produced on channel $c \in O$ in the next time interval.

I associate a stream processing function F_s with a state machine that is given by the transition function Δ using the following definition. More precisely, I associate a time guarded function F_s with every state $s \in$ State as it is defined by the following equation:

$$\begin{aligned}F_s(x) = \{y \in \vec{O}: &\\ &(\exists\, i \in \vec{I}^*, o \in \vec{O}^*, s' \in \text{State}, x' \in \vec{I}, y' \in \vec{O}:\\ &\qquad \overline{y} = o\hat{\ }\overline{y'} \wedge \overline{x} = i\hat{\ }\overline{x'} \wedge \Delta(s, i).(s', o) \wedge y' \in F_{s'}(x')) \vee\\ &(\forall\, i \in \vec{I}^*: i \to \overline{x} \Rightarrow \forall\, o \in \vec{O}^*, s' \in \text{State}: \neg\Delta(s, i).(s', o))\}\end{aligned}$$

If the state transition function contains cycles then the definition of F_s as given above is recursive. In this case, I cannot be sure that by the equation above the behaviour F_s is uniquely specified. Therefore I define F_s to be the largest (in the sense of elementwise set inclusion) time guarded function that fulfils this equation.

Along these lines a precise treatment for sophisticated concepts like priorities and spontaneous transitions due to our carefully chosen semantic model that includes time is possible. Without an explicit notion (at least on the semantic level) of time a proper semantical treatment of priorities or of spontaneous reactions is difficult or even impossible. To demonstrate the use of state transition diagrams for the description of the behaviour of components I give a first simple example.

Example: Alternating Bit Protocol

As an example I consider the alternating bit protocol. It has the distribution structure given in Figure 4.

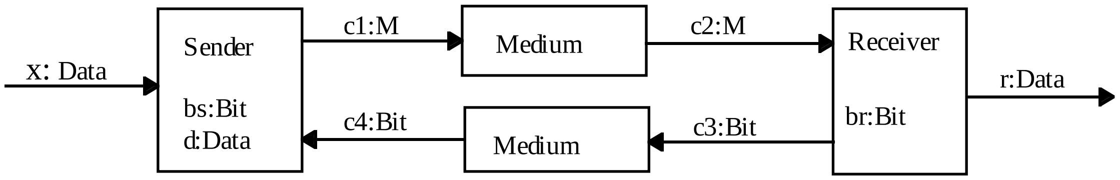

Figure 4 Data Flow Net of the Alternating Bit Protocol

I use the following sorts:

sort Data
sort M = m(Data, Bit)

The state transition diagram of the component Receiver is very simple. It is given by Figure 5.

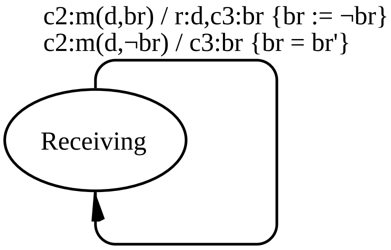
c2:m(d,br) / r:d,c3:br {br := ¬br}
c2:m(d,¬br) / c3:br {br = br'}
Receiving

Figure 5 State Transition Diagram of the Receiver

The syntactic interface and the state space of the sender and the receiver is indicated by the data flow node given in Figure 4. The sender has a more sophisticated state transition diagram, since it has to react. It is shown in Figure 6.

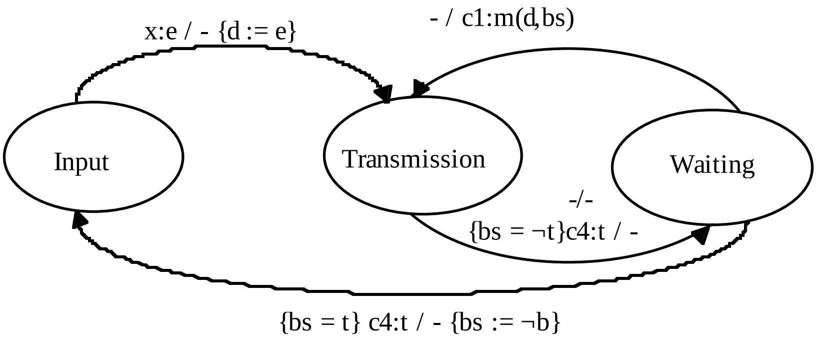

Figure 6 State Transition Diagram of the Sender

This state transition diagram contains a spontaneous transition. It is needed for the case where messages are lost by the transmitter. This corresponds to a soft timeout.

The component Transmitter is very simple. Figure 4 gives it as a data flow node. Figure 7 shows the state transition diagram of the transmitter. It does not include any fairness assumptions. Fairness assumptions can be included by prophecies in the state space or by additional equations for the transmitter.

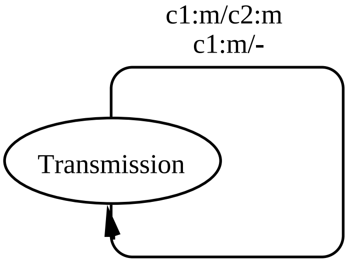

Figure 7 State Transition Diagram of the Transmitter

7 DISTRIBUTED SYSTEMS AND THEIR DESCRIPTION

In this section I introduce a mathematical notion of distributed systems. An interactive distributed system consists of a family of interacting components often also called *agents* or *objects*. These components interact by exchanging messages on channels by which they are connected. A structural view onto a distributed system consists of a network of communicating components. Its nodes model components and its arcs communication lines (channels) on which streams of messages are sent. A glass box view of a system component can be represented by a distributed system or by a state machine. In the first case I speak of a *composed distributed system* and in the later case I speak of a *nondistributed system*. Figure 4 shows the data flow representation of a simple example of a distributed system.

I model distributed systems by data flow nets. Let N be a set of identifiers for components (each of which is represented by a data flow node) and O be a set of output channels. A distributed system (ν, O) with the syntactic interface (I, O) is given by a mapping

$$\nu: N \rightarrow \text{Ç}$$

that associates with every node a component representing a behaviour (an interface behaviour). Here I denotes the set of input channels, that can be determined by those input channels of the components of the system that do not have a source (are not output channels of a component of the system), and O denotes the set of output channels of the system.

I get an abstraction of a distributed system to its black box view by mapping the distributed system onto a component behaviour $F \in \text{Ç}[I, O]$. Every data flow net defines a *black box view* given by a component behaviour F via the following specification:

$$F(x) = \{y|_O: y|_I = x \wedge \forall\, i \in N: y|_{Out(\nu(i))} \in \nu(i)(y|_{In(\nu(i))}) \}$$

For a function $g: D \rightarrow R$ and a set $T \subseteq D$ the function $g|_T: T \rightarrow R$ denotes the restriction of the function g to the domain T. The formula above essentially is based on the idea that the output of the net is the restriction of a fixpoint for all the channel equations induced by the network.

A distributed system can be described graphically by a data flow network (called CD for communication diagram in GRAPES, see GRAPES-Referenzmanual (1990), block diagram in SDL, see SDL (1988)) which is a directed graph. This way the structure of distributed systems can be represented.

8 PROCESSES

A process is a family of events which are instances of actions that are in some causal relationship. In the case of interacting systems an event is triggered by a number of messages received. When it is carried out it results in sending a number of messages that may cause further actions. If in a process an event a1 is directly causal for an event a2 there must exist a message sent from event a1 to event a2.

Each instance of sending or receiving of a message is called an *event*. Events that are caused by the environment are called *external* events. All other events are called *internal* events. An internal event the receiver of which is the environment is called an *output* event. Special events are *time* events. They can be understood also as messages that are sent by a timer.

In our model of a distributed system a process is represented by an acyclic data flow net (ν_p, O_p) and a valuation function

$$\eta: Chan((\nu_p, O_p)) \rightarrow M$$

that associates with all of the arcs of the net exactly one message.

An individual history of a system behaviour (also called a run of a system) can be described by the set of events (exchanged messages) and their causal relationship. It is represented by a process. In software engineering, the concept of a process for illustrating individual cases of interaction is found for instance in SDL (see SDL (1988)) called *message sequence charts* and recently developed into an independent standard or in Objectory (see Booch (1991)) called *use cases*. I can illustrate the co-operation between the components by message sequence charts like it is shown by Figure 8.

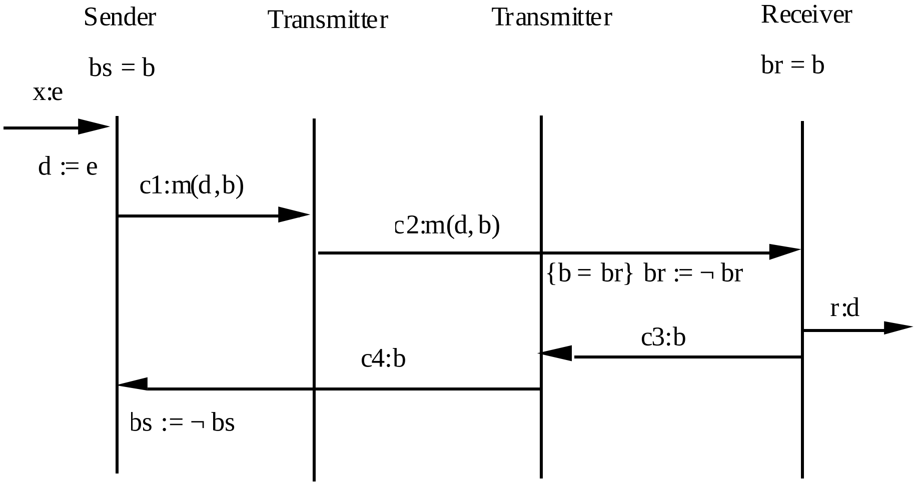

Figure 8 Scenario: Successful Transmission - Successful Acknowledgement

The interaction can also be described by a process diagram as found in the GRAPES-Referenzmanual (1990) as it is given in Figure 9.

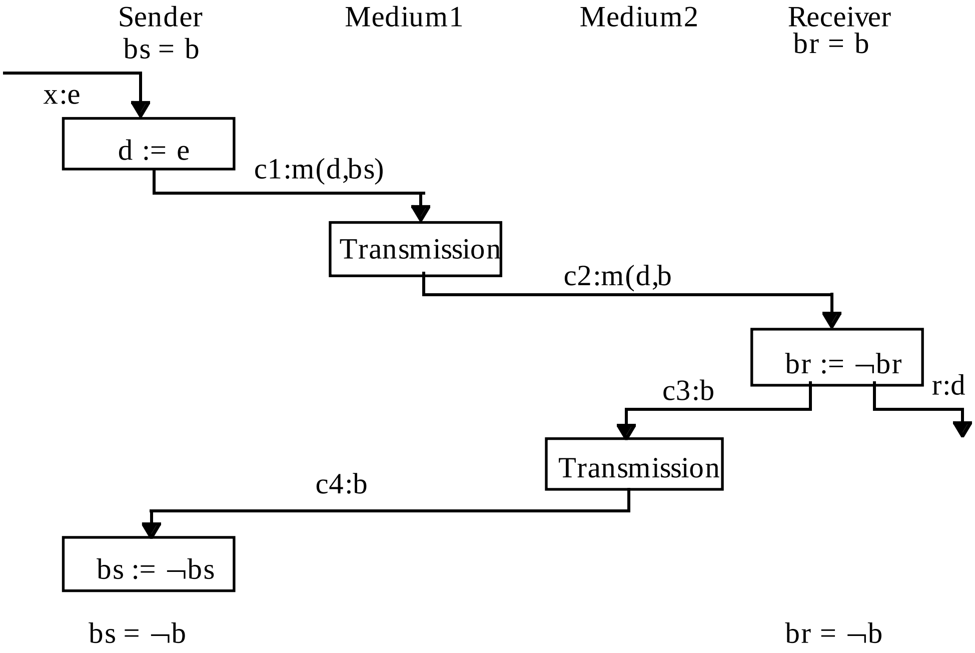

Figure 9 Process Diagram

Processes of interactive systems are here represented by acyclic data flow graphs where each node stands for exactly one action and each arc stands for one event of a message or signal transfer from one action to another. So each action in the process is represented by exactly one node and each event is represented by exactly one arc.

9 DEVELOPMENT TECHNIQUES

A development method gives hints and suggestions, how to use FDTs in a systematic way in system development. In software engineering the development is carried out by going through development phases. I speak of a *development process model.* Mathematical techniques can help to characterize the required relation between the various views and documents in a development process. They help to understand whether a particular proceeding is appropriate and how the interaction between the various documents and views can be formalised and supported by methods and tools.

In software engineering, systems are described by a number of complementing views and on different levels of abstraction. Of course, we need to integrate these views. Furthermore we need clear mathematical relations between the different levels of abstraction. This is closely related to refinement notions.There are the following two concepts of refinement:

- *property refinement*: In property refinement a system is developed by adding further properties (requirements) and further system parts (enriching the signature). The basic mathematical notion of property refinement is logical implication (with respect to the logical properties) or set inclusion (with respect to the set of models).
- *representation refinement* (a special case is *data refinement*): In representation refinement I change the representation of a data model or of the states and messages of a system model. This can be done with the help of a function relating the two models.

For data models representation refinement was studied extensively over the last 25 years after stimulating papers by Hoare. For models of system components such studies have emerged only more recently. For the system components I deal with three concepts of refinement

- property refinement
- interface refinement
- glass box refinement

By property refinement I do not change the syntactic interface of a component but add properties to the specification. This reduces the set of possible output histories.

In interface refinement I also may change the syntactic interface, but also then I insist on a well-specified relationship between the behaviour of the refined component and the original one. In glass box refinement I replace the interface view by a glass box view. A glass box view may be provided either by a state machine or by a distributed system. The black box behaviour associated with these has to be a property refinement of the original system.

For our concept of a system model *property refinement* is very simple. A component with interface view

$\hat{f} \in$ Ç[I, O]

is called a *refinement of a component*:

$f \in$ Ç[I, O]

if for all input streams $x \in \vec{I}$ we have:

$$\hat{f}(x) \subseteq f(x)$$

I write then

$$\hat{f} \subseteq f$$

For a systematic use of refinement concepts, *compositionality* (which is the mathematical version of *modularity*) of refinement is essential. Compositionality means that if we replace a component of a system by its refinement we obtain a refinement of the overall system.

An *interface refinement* changes the syntactic interface of a component, but provides a precise interpretation of the behaviour of the refined component as behaviours of the given one. I introduce what is called an *upwards simulation* in the literature. I consider two components with different syntactic interfaces:

$$f \in \text{Ç}[I, O],$$

$$\hat{f} \in \text{Ç}[\hat{I}, \hat{O}].$$

If I want to consider the behaviour $\hat{f}$ as an interface refinement of the behaviour f, I have to interpret all computations of $\hat{f}$ as behaviours of f. Therefore I introduce the concept of a *channel history refinement*. Let Z and $\hat{Z}$ be sets of channels. A pair of behaviour functions

$$A \in \text{Ç}[\hat{Z}, Z], \qquad R \in \text{Ç}[Z, \hat{Z}],$$

is called a *channel history refinement* if we have:

$$R ; A = Id$$

where Id is the identity function and „;“ is usual functional composition (relational product). For an interface refinement I need two channel history refinements:

$$A_I \in \text{Ç}[\hat{I}, I], \qquad R_I \in \text{Ç}[I, \hat{I}],$$

$$A_O \in \text{Ç}[\hat{O}, O], \qquad R_O \in \text{Ç}[O, \hat{O}],$$

such that

$$R_I ; \hat{f} ; A_O \subseteq f$$

So for every input of the component f a computation of f is represented by a computation of the component $\hat{f}$.

I do not have to say much about glass box refinement. In the sections on state transition systems and distributed systems I have specified how to derive a black box view from a state transition description or from a description of a distributed system. In glass box refinement I simply use this definition by reversing the direction of development. Starting from a black box description I work out a state transition description or a description of a distributed system the black box view of these is a property refinement of the original black box description.

A distributed system is refined by refining its components. For a component I may use a property refinement which always leads to a property refinement of the distributed system. I may refine a component into a state transition machine or a distributed system. Also this leads into the refinement of the distributed system. Of course I may also refine the communication histories for the internal channels by an interaction refinement. Also this can be done in a modular way.

Since in our approach, processes are only a special case of distributed systems the concepts of component refinement carry over to process refinement. We can apply all three concepts of refinement to processes such as property refinement for actions, interface refinement and the glass box refinement that allows us to replace an action by a process or to describe an action by a state machine.

The development of systems can be understood as a joint development of the process, the data, the behaviour and the structural model. Each of these views is only worked out to a level of detail regarded appropriate.

The different views developed in systems modelling finally have to be integrated into a consistent system description. View integration can be made on a purely mathematical level by joining together all the mathematical structures derived from the various description methods. Another possibility is to understand every description method as a logical statement about the system. Then all the logical statements provided by the description can be composed into one big axiomatic specification. Then consistency of a description coincides with logical consistency. Here axiomatic specification techniques are extremely helpful. They allow to translate all views of a system into a set of axioms about a system. This technique is successfully applied by Hußmann (1995). There the practical development method SSADM is defined by translating it into axiomatic specifications.

10 CONCLUSIONS

We are at an exciting stage in system engineering and the integration of formal methods. A lot of the theoretical work required for the foundations has been done. At the same time FDTs have found their way into practical use. What is needed is scientific consolidation of FDTs by an experimental integration of the theoretical results and the practical experiences gained by their application in practice.

In the SysLab-project at the Technical University of Munich we try to follow this line and make a significant effort to work out scientific foundations for software engineering and to integrate formal methods into pragmatic software engineering methods.

Acknowledgement

The thoughts presented above have benefited greatly from discussions within the SysLab team, the IFIP working group 2.3, with software engineers from BMW, ESG, Siemens, Siemens Nixdorf, Digital and many others.

References

Beeck, M. v. d. (1994) A Comparison of State Charts Variants. In: H. Langmaack, W.-P. de Roever, J. Vytopil (eds): Formal Techniques in Real Time and Fault-Tolereant Systems. Lecture Notes in Computer Science 863, 128-148.

Berry, G. Gonthier, G. (1988) The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. INRIA, Research Report 842

Booch, G. (1991) Object Oriented Design with Applications. Benjamin Cummings, Redwood City, CA.

Broy, M. (1991a) Towards a Formal Foundation of the Specification and Description Language SDL. Formal Aspects of Computing 3, 21-57.

Broy, M. (1995a) 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.

Broy, M. (1995b) Mathematics of Software Engineering. Invited talk at MPC 95. In: B. Möller (ed.): Mathematics of Program Construction, July 1995, Kloster Irsee, Lecture Notes of Computer Science 947, Springer, 18-47.

Broy, M. (1995c) Mathematical System Models in Software Engineering. J. van Leeuwen (ed.): Computer Science Today. Lecture Notes of Computer Science 1000, 292-306.

Broy, M. (1996) Abstract Semantics of Synchronous Languages: The Example ESTEREL. Unpublished Manuscript

Broy, M., Facchi, C., Hettler, R., Hußmann, H., Nazareth, D., Regensburger, F., Slotosch, O. and St∅len, K. (1993) The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I/II Technische Universität München, Institut für Informatik, TUM-I9311 / TUM-I9312.

Facchi, C. (1995) Methodik zur formalen Spezifikation des ISO/OSI-Schichtenmodells. Dissertation, Fakultät für Informatik, Technische Universität München.

GRAPES-Referenzmanual (1990), DOMINO, Integrierte Verfahrenstechnik. Siemens AG, Bereich Daten-und Informationstechnik.

Harel, D. (1987) Statecharts: A Visual Formalism for Complex Systems. Science of Compute Programming 8, 231-274.

Hettler, R. (1994) Zur Übersetzung von E/R-Schemata nach SPECTRUM. Technischer Bericht TUM-I9409, Technische Universität München.

Hinkel, U. (1995) Einbettung von SDL in Logik (Teil 2). Technische Universität München, Institut für Informatik, interner Bericht

Hußmann, H.(1995) Formal Foundations for SSADM. Technische Universität München, Fakultät für Informatik, Habilitationsschrift.

SDL (1988) Specification and Description Language (SDL), Recommendation Z.100. Technical report, CCITT.

Wirsing, M. (1990) Algebraic Specification. Handbook of Theoretical Computer Science, Vol. B, Amsterdam: North Holland, 675-788.