Mathematical Methods in System and Software Engineering*

Manfred Broy
Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract

Today, there is still a remarkable gap between the techniques and methods used in practice in software engineering and the formal techniques worked out and advocated by academics. Our goal is to close that gap and to bring together the pragmatic and mostly informal ideas in systems and software engineering used in practice and the mathematical techniques for the formal specification, refinement and verification of software systems. In practice, software engineers are used to work with

- a development method that describes the development process in detail and
- description formalisms that describe the system under development; these descriptions are often annotated diagrams.

The development process is often supported by CASE (Computer Aided Software Engineering) tools.

We present a mathematical, scientific basis for system and software engineering. In its core, there is a mathematical model of a system and formal techniques to describe it. We outline representative examples of diagrammatic description techniques as they are used in software engineering approaches in practice and show how they are formally related to the system model. These description techniques include in particular

- data models,
- process models,
- structure and distribution models,
- state transition models,
- interface models.

We define a translation of the description techniques into predicate logic. This allows us to combine techniques of formal specification and verification with pragmatic system description techniques. We show how to develop systems with the help of these description techniques in refinement steps. By this, we demonstrate how software engineering methods can be backed up by mathematics. We discuss the benefits of such a mathematical and scientific foundation. These benefits go far beyond the benefits of the formal methods for the specification and verification of software.

---

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

1. Introduction

It is widely accepted by now that the development of software is an engineering discipline. To underline this we speak of *software engineering* and also of *system engineering* in cases where the software is embedded into a system context. Typically, all engineering disciplines of today are based on theoretical and, in particular, on mathematical foundations. These theoretical foundations are used as a basis for a deeper understanding of the engineering task, for a body of rules, procedures and engineering processes. There cannot be any doubts that software engineering has and needs its own theoretical and mathematical foundation like any other engineering discipline.

Software engineering, in practice, deals with the development of large and often complex information processing systems. This includes techniques for the description of requirements and systems in the disparate development phases. The main goals of software engineering are a good quality of the engineering process and its results, the timely delivery of the end procuct, and high productivity. In this paper we ignore most of the economical aspects and sketch how a mathematical basis of the technical aspects of system and software engineering may look like.

1.1 Formal Methods

Starting with the pioneering work of Backus, Bauer, Samelson, McCarthy, Petri, Strachey, Scott, Dijkstra, Floyd, Hoare, deBakker, Reynolds, Milner, the VDM group and many others formal methods in software development have been extensively investigated over the last 25 years. Today formal methods are a well-established subject in academia.

After denotational semantics, axiomatic specifications and numerous logical calculi are available, many of the theoretical problems for modelling software engineering concepts by mathematical and logical techniques are solved (see also [Abrial 92]). More work is needed, however, along the lines of [Jones 86] and [Guttag, Horning 93] to make the theoretical work better accessible to software engineers working in practice. The fact that theoretical foundations are sufficient already today to cover the foundations of pragmatic methods is demonstrated, for instance, by [Hußmann 95] which provides a complete formalization of the method SSADM by translating it into SPECTRUM. SSADM is a pragmatic requirement engineering method used as the UK government's standard method for carrying out systems analysis and design stages of information technology development projects. SPECTRUM is an axiomatic specification language (cf. [SPECTRUM 93]).

How many basic problems still have to be studied and how much foundational work is still needed and stimulated by practical problems can be seen, for instance, by the concept of state charts (see [Beeck 94] for a long list of open problems) for describing embedded process control systems. Statecharts (see [Harel 87]) are - also due to the fact that they are supported by a commercial CASE tool - by now widely used in industry.

1.2 From Theory to Practice

Thirty years ago, for many concepts used in computing practice, a theoretical foundation was completely missing. This is no longer true today. A solid body of foundational work is available by now, giving principal answers to many of the demanding questions of theoretical foundations of software engineering. We have denotational semantics, axiomatic specification techniques for handling data structures, functions and statements, we have assertion calculi, logical calculi for parallel and distributed systems and various logical concepts for dealing with interactive systems.

The next consequent step is to increase the experimentation and the transfer of these theoretical results to engineering practice. Yet, this needs the good will and professional attitude of both the practical software engineers and the theoreticians. This means that also the theoreticians are requested to undertake serious efforts to get a better understanding of the needs of software engineers working in practice and of their advanced engineering concepts.

Development needs methodological guidance. In addition to the specification notation and logical calculi and maybe a few small, hopefully illustrative case studies, more methodological guidance is required to bring theoretical ideas closer to practical use. It is not realistic that everyone in practice develops her own ideas how to use the methods provided by theoreticians in a systematic appropriate manner.

1.3 Elements of Software Engineering

The development of large software systems involves a large number of people with quite different skills. Its goal is the construction of a technical solution to a client's problem. Accordingly, software engineering is a discipline that involves organisational and management aspects as well as technical aspects. A decisive issue is the cost management.

Software engineering, like many other engineering disciplines in their early phases, has developed a rich and often confusing plenty of proposals for overcoming the difficulties of organising large development processes. Here a major issue are economical aspects. These include among others

- team organisation and management,
- cost and schedule prediction, statistical models to evaluate the quality of software,
- development process organisation and integration (phases, mile stones, documentation),
- integration with existing software components,
- tool support.

It is naive to assume that managers responsible for large projects would switch to new more formal development methods for which these questions are not convincingly answered.

Of course, it is a long way to provide a complete mathematically well-founded software development method, but only when we start to work in this direction there might be a chance to transfer theoretical results more effectively into practice. When working towards such a method, we will discover many challenging theoretical problems. Examples are the mathematical capture of the concepts of software

architectures, design patterns, and the formalization of existing pragmatic description methods.

We do not treat economical aspects in the following at all and concentrate only on technical aspects. These comprise:

- development process organisation (phases, mile stones, documentation),
- formulation of strategic goals and of constraints,
- modelling, description, specification,
- quality guarantee,
- integration with existing software components, reuse,
- documentation,
- tool support.

Although we will concentrate in the following mainly on the technical aspects, we should keep in mind that there is a close relationship between the technical and the management aspects. The best technical solution is worthless if it is too costly or if the product cannot be finished in time when working with it. Moreover, most of the management and planing cannot be carried out without a deep understanding of the technical tasks.

Whenever it is necessary to assess the role of technical aspects with respect to management aspects, we will refer to them. We find the closest connection between management and technical issues in the organisation of the development process, being part of what is called *process model*. A central notion in software engineering is the development method[1]. A method in software engineering comprises description techniques in its syntax, semantics and pragmatics, rules for working with these techniques, development techniques and general principles.

1.4 Overall Organisation of the Paper

A software system is represented by a set of descriptions in the form of textual or graphical documents, and, at the same time, it is a product that can be brought to life and then show a complex dynamic behaviour. The goal of the software development process is a model of the application described by formalisms for which an efficient execution is available. Therefore, like no other engineering discipline, software engineering deals with models, description formalisms and modelling techniques. In the following we want to carefully distinguish between

- *mathematical models*: the mathematical structures forming the semantical conceptual model associated with a system or a software description formalism,
- *description techniques*: the notations of the descriptions, given by the syntax, graphics, or tables used in the documentation of a software system,
- *modelling techniques*: the activities and methods mapping, representing, and relating real life aspects of applications by using software description techniques.

Of course, these three aspects are closely related. Nevertheless, we want to carefully perceive these distinctions in the following, since they are of major relevance for the understanding and foundation of software engineering. It is, in particular, important to distinguish between the description formalisms and the modelling idea for systems.

[1] We distinguish between method and methodology. Methodology is the systematic study of methods.

This paper is an attempt to give a comprehensive mathematical foundation to software engineering formalisms and methods in the form of mathematical models and relations between them. These particular mathematical models should be understood rather as an instance of what we are aiming at and not as the only way to provide a mathematical foundation. We want to demonstrate, however, that a comprehensive family of models and well-founded description techniques for software engineering can be provided.

The paper is organised as follows. In section 2 we give a survey of the most important notions we deal with. In section 3 - 7 we then deal with the notions of a data model, a process, a component, a state machine and a distributed system and give mathematical denotations for them. In section 8 we deal with development methods, with refinement, and development process models. We conclude after a short section on implementation issues and tools.

Throughout this paper we use as our running example a simple adaptation of [Broy, Lamport 93] (see also [Broy, Merz, Spies 96]). It deals with a simple memory and a component to access it. The storage is faulty. This means that we can read or write the storage but as a result of reading or writing we may get failures. We assume that those failures are indicated by failure messages which allow us to do some exception handling either by indicating to the environment that something went has gone wrong or by retrying. We specify such an unreliable memory and components that organise the access to the unreliable memory. We use the faulty store as a running example for the specification of the data models, the process models, the component models and the distributed system as well as the state machines.

2. Mathematical Models and System Description

In the following sections we define mathematical models and description techniques for systems and system aspects. In this section we disucuss the role of formal methods and mathematics in systems engineering.

2.1 System Aspects and Views

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. These include:

- data models,
- system component models,
- distributed system models,
- state transition models,
- process models.

For each of these aspects of a system a mathematical model is defined in the following, consisting of a syntactic and a semantic part. We use these models as a basis for giving meaning to more pragmatic description formalisms.

2.2 Description Formalisms in Software Engineering

In software engineering it is necessary to describe large information processing systems and their properties. Of course, this cannot be done by only providing one description formalism or one single document. Therefore software engineering methods suggest several complementing description formalisms and documents. Each of these documents describes particular views on the system. Putting these views together should lead to a consistent comprehensive system model. We speak of *view integration.*

Integration of description formalisms and their mathematics is an indispensable requisite for more advanced software development methods. In the following, we discuss some of the more widely used description formalisms in software engineering and their mathematics.

Description formalisms are used to represent models like programming languages are used to represent algorithms. It is helpful therefore to distinguish between description formalisms and modelling techniques. A description formalism is a textual or graphical formalism, such as for instance a diagram or a table, for describing systems, system parts, system views and system aspects. A model is a mathematical structure to be used as an image of certain aspects of a real life system. Description formalisms allow us to represent such models and views as well as their properties by concrete syntactic means. Mathematical semantic models are mathematical structures that are used to give a precise meaning to description techniques.

Practical software engineers tend to overemphasise the importance of description formalisms in terms of syntax while theoreticians often underestimate the significance of the notation they use. In the engineering process, notation, be it graphics, tables, or texts, serves as a vehicle for expressing ideas, for analysis, communication, and documentation. To provide optimal support for these purposes both the presentation of the descriptions and the mathematical models behind them have to be well chosen. Since, depending on the purpose of the modelling and the education and experience of the user, the effectiveness of the presentations may be quite different, it is generally wise to have several presentations for the same description. Examples are text, diagrams, and tables.

When analysing and describing systems, we distinguish between static and dynamic aspects. Static aspects of an information processing system are those that do not change over the life time of a system. Dynamic aspects have to do with behaviour. We can often use the same description formalisms for the static and for the dynamic aspects.

Dynamic aspects of components can be described either axiomatically, property-oriented, or in operational terms. Operational descriptions tend to be less abstract. Therefore, nonoperational description often are better suited in the early phases of development. Both property-oriented descriptions as well as operational descriptions can rigorously based on logic, of course. To improve the redability, we can also give graphical as more illustrative formalisms for describing properties of systems. For such descriptions, a translation into logical formulas should be provided.

2.3 The Role of Description Formalisms in the Development Process

Description formalisms and the underlying semantic models form the basis for formulating and documenting of the results of the development process. We may

understand the development process as the working out of documents describing the requirements and the design decision of the system in more and more details, adding implementation aspects until a desired implementation level is reached.

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 each of the various aspects of a system. A proper relationship between these description techniques and a foundation of them are one of the goals of the mathematics of software engineering.

2.4 The Role of Logic

A very universal and precise discipline for the description of all kinds of properties and aspects of a system is mathematical logic. A logical formalism provides a formal syntax, its 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.

Properly defined description formalisms also have a formal syntax and a mathematical semantics. Often, however, deduction rules are not provided. Nevertheless, for most description formalisms we may define transformation rules for manipulating them. In contrast to logical formalisms, the description formalisms of software engineering are not especially designed for the formal manipulation by deduction and transformation rules.

Description techniques used in practice often do not have a mathematical semantics and not even a proper informal description of their meaning.

In any case, 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.

3. Data Models and their Specification

In this section we introduce a mathematical model for data structures on which specification techniques can be based. Then we introduce perceptual description techniques for data structures and relate them to the mathematical model.

3.1 Mathematical Data Models

Data models are needed to represent the information and data structures involved in an application domain. They are also used to provide computing structures for representing the internal data of information processing systems. In general, data models capture mainly the structure of the data and their relationship, but not their characteristic operations. These, of course, should by an integral part of every data model. Therefore we understand a data model always as family of data sets named by sorts together with their relationships and the basic characteristic operations and functions.

Families of sets and according operations are treated in mathematics in algebra. From a mathematical point of view, a data model is a heterogeneous algebra. Such an algebra is given by a family of carrier sets and a family of functions. More technically, we assume a set S of sorts[2)] (often also called types or modes) and a set F of constants 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. We assume that the set of sorts S contains besides basic sorts (such as Bool, Nat, etc.) also tuples of sorts as well as functional sorts and even polymorphic sorts (see [SPECTRUM 93] for details). Both sets S and F provide only names. The pair (S, F) together with 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 on a data model.

In every algebra A of the signature (S, F) we 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. An algebra gives meaning to a signature and can be seen as the semantics of a data model. It is typical for mathematical structures modelling information processing concepts that they include static (syntactic) parts such as name spaces (in the algebraic case the signature) and semantic parts (in the algebraic case carrier sets and functions).

Data models are used to capture various aspects of software systems which often go much beyond pure data and information structure aspects. In other words, they may be used also to represent dynamic aspects of system behaviours or system structures and not only the static structures of data sets and selector functions. That such aspects can also be captured by data models is not surprising. Following our definitions, data models can be seen as algebras and algebras are a very general mathematical concept. All kinds of mathematical structures and mathematical models of systems can be understood as heterogeneous algebras. Nevertheless, we suggest to use the general concept of algebras for data models only. In the following we give more specific mathematical structures for other system aspects.

3.2 Description of Data Models by Sort Declarations

In the data view, we fix the relevant sorts and their characteristic functions for the system we want to describe or implement. In addition, we can describe the system states. This can be done by sort declarations as we find them in programming

---

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

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 their components.

In our example we use a sort CalMem and fix the data attributes of it by giving their sorts. The sort CalMem is described by the following sort declaration:

Sort CalMem = put (i: Location, d: MemVals) | get (i: Location)

In a sort declaration we introduce a sort (in the example above the sort CalMem) and describe it by a sort expression. The sort expression is formed of enumeration sorts, records and variants. An enumeration sort is given by an expression of the form

$\{a_1, ..., a_n\}$

and introduces the identifiers $a_1, ..., a_n$ as constants, which are the constructors of the enumeration sort.

A record sort has the form

con (sel_1: M_1, ..., sel_n: M_n)

where M_1, ..., M_n are sorts. It introduces con as a constructor function and $sel_1, \ldots, sel_n$ as selector functions. Here, we assume the convention that we use the sort identifiers as selectors, if no selectors are mentioned explicitly.

A variant sort has the form

$R_1 \mid ... \mid R_n$

where $R_1, ..., R_n$ are record or enumeration sorts. The constructors for these sorts are used as constructors and discriminators for the variant sort.

3.3 Description of Data Models by Logic

Data models can be described by the logical properties of the functions involved. Then we speak of the axiomatic or of the algebraic specification of data structures. The techniques of algebraic specification is in the meanwhile well-understood. Therefore we just give a simple example and refer to [Wirsing 90] for an overview and to [SPECTRUM 93] for an instance of a particular algebraic specification language.

We only give two simple examples. We start with the polymorphic specification of the algebra of finite sets.

SPEC SET =

{ **sort** Set α,

ø : Set α,
$\_\cup\{\_\}, \_\setminus\{\_\}$: Set α, $\alpha \rightarrow$ Set α,
$_=$ø : Set $\alpha \rightarrow$ Bool,
$\_\in\_$: α, Set $\alpha \rightarrow$ Bool,

Set α **generated_by** ø, $\cup$,

ø =ø,
$\neg(s\cup\{x\} =$ø),

$\neg(x \in \emptyset)$,
$x \in s \cup \{x\}$,
$x \in s \Rightarrow x \in s \cup \{y\}$,
$x \neq y \Rightarrow x \in s \cup \{y\} = x \in s$,

$\emptyset \setminus \{x\} = \emptyset$,
$(s \cup \{x\}) \setminus \{x\} = s \setminus \{x\}$,
$x \neq y \Rightarrow (s \cup \{y\}) \setminus \{x\} = (s \setminus \{x\}) \cup \{y\}$,

$(s \cup \{x\}) \cup \{y\} = (s \cup \{y\}) \cup \{x\}$,
$x \in s \Rightarrow s \cup \{x\} = s$ }

The sorts such as enumeration sorts, records and variants used in sort declarations given above can be schematically translated into algebraic specifications.

As a second example we define a simple object model by algebraic techniques. The object model describes for given classes α and given attributes for each class the object model in a polymorphic style. The sort Store denotes the set of object stores and the set Obj α denotes the set of object identifiers for the class α. More precisely, α is a record sort of the attributes of the class.

OBJECT_MODEL = {

sort	Store, Obj α,	
emptyStore :	Store,	*empty store*
update :	Store, Obj α, $\alpha \rightarrow$ Store,	*update of an object*
newObj :	Store, $\alpha \rightarrow$ Obj α,	*creation of an object*
newObjstore :	Store, $\alpha \rightarrow$ Store,	*allocation of storage for a new object*
valid :	Store, Obj $\alpha \rightarrow$ Bool,	*test, if object id is declared for a store*
deref :	Store, Obj $\alpha \rightarrow \alpha$,	*dereferencing*

Axioms:

For the empty store:

valid(emptyStore, r) = false,

For selective update:

valid(σ, v) $\Rightarrow$ valid(update(σ, v, a), r) = valid(σ, r) $\wedge$
deref(update(σ, v, a), r) = **if** r = v **then** a **else** deref(σ, r) **fi**

For object storage allocation:

valid(σ, newObj(σ, a)) = false,
valid(newObjstore(σ, a), r) = (valid(σ, r) $\vee$ r = newObj(σ, a)),
valid(newObjstore(σ, a), r) $\Rightarrow$
deref(newObjstore(σ, a), r) = **if** r = newObj(σ, a) **then** a **else** deref(σ, r) **fi**
}

Here Obj α is a polymorphic sort. This means that for every sort M we may form the sort Obj M.

3.4 Data Models of Component States

Data models can be used to model the state of a component. A mathematical notion of a component will be given in the following section.

Given a signature (S, F) as a basis, we 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, as well as the symbols in F, a sort.

A widely used technique to describe state spaces are entity/relationship-diagrams. In an entity/relationship diagram we find notations and symbols as shown in Fig. 1.

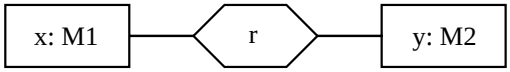

Fig. 1 Two Entities x and y Connected by One Relationship

The diagram as given in Fig 1 is used to express that x, r, y are state attributes. The shape of the graphical symbols indicates that x and y are entities with the sorts as shown. These nodes are equivalent to the declarations

x: Set M1

y: Set M2

The shape of the symbol of r indicates that r is a relationship. This corresponds to the declaration

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, we have to add invariants such that the relation r contains in every state only pairs (a, b) with elements a that are in x and elements b that are in y, respectively.

Formally, this is expressed by

$(a, b) \in r \Rightarrow a \in x \wedge b \in y$

We also allow to include simple attributes in the state space that correspond to program variables.

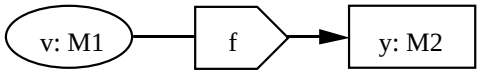

Fig. 2 Functional Relationship

We write the diagram shown in Fig. 2 to express that v is an attribute

v: M1

and f is a function attribute (a program variable)

f: M1 $\rightarrow$ M2

that maps v onto an element of the entity y: Set M2. If we replace the attribute by an entity, this expresses that every element in the set is mapped by f onto an element of y. Again we assume an invariant:

$f(v) \in y$

Data models can thus be used to describe the information structure of an application and the state space of components. In our example system, we use a component Store. The data view is described by an E/R-model. It is based on the sorts given in a table of sort declarations in Tab. 1. Fig 3 gives a simple example of a data structure diagram and an E/R-diagram.

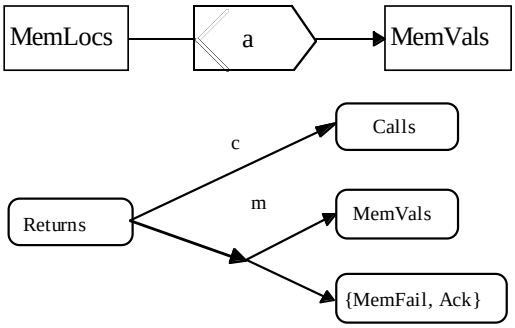

Fig. 3 Data Sort Diagram

With this data view of the state of the component Store the behaviour of the component Store can be made more precise later. Each action of a component may change some of its attributes. We come back to this issue in section 5 on the description of behaviours.

Tab. 1 Table of Data Sorts and Their Declaration

MemLocs		memory locations,
MemVals		memory values,
PrIds		identifiers for processes,
CalMem	= put (i: MemLocs, d: MemVals) | get (i: MemLocs)	memory calls
RetVals	= MemVals | {MemFail, Ack}	return values,
RetMem	= ret (c: CalMem, rv: RetVals)	return messages of memory,
Returns	= ren (c: Calls, m: RetVals)	return messages of the driver,
Calls	= ca (pi: PrIds, mc: CalMem)	calls for the driver.

Graphical representations and diagrams can be used as an illustrative, but nevertheless fully formal description technique, if 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 careful translation of E/R-concepts into axiomatic specifications is given by [Hettler 94]. Hettler distinguishes between E/R-techniques for modelling static aspects (the static data model) and dynamic aspects of systems (such as the states of a data base). The sort MemLocs is described by a record sort listing its attributes. In the dynamic view, a program variable of sort Set MemLocs is associated with the entity MemLocs.

Axiomatic and algebraic specification techniques as well as E/R-diagrams provide powerful and very general description formalisms. They allow us to describe data

sorts and their characteristic operations in an abstract way not biased by the choice of a concrete representation. They follow the principle of information hiding of implementation information in a consequent manner.

4. System Components

In this section we introduce a mathematical concept of a system component and techniques for describing components. A component is considered in practice as a black box that is an encapsulation of related services according to a published specification. Our mathematical model of a component is consistent with this informal description.

4.1 Component Models: Interface Models by Stream Processing Functions

We are interested in system models that allow us to represent systems in a modular way. We think of a system as being composed of a number of subsystems that we 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 clear cut interface. Through its interface it is connected with its environment. In this section we introduce a simple, very abstract mathematical notion of a system component.

For us, 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.

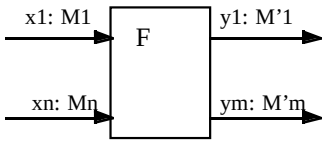

Fig. 4 Graphical Representation of a Component as a Data Flow Node with Input Channels x1, ..., xn and Output Channels y1, ... , ym and their Respective Sorts M1, ... , Mn and M'1, ..., M'n

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 are only interested in the interface of a component with its environment. For this we have to describe the causal dependencies between the input and the output messages. In a glass box view we are interested in the internal structure of a component, which can either be given by its local state space together with a state transition relation or by its decomposition into subcomponents. We 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, we use just one set M of data sorts for messages on the channels in the sequel in order to keep the presentation simple. A graphical representation of a component with its syntactic interface is shown in Fig. 4.

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 set of infinite sequences over the set of messages M. By $\hat{}$ we denote the concatenation of sequences. The set M^∞ can be understood to be represented by the total mappings from the natural numbers $\mathbb{N}$ into M. Formally we define the set of timed streams by (we write S^∞ for the function space $\mathbb{N}^+ \to S$ and $\mathbb{N}^+$ for $\mathbb{N} \setminus \{0\}$)

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

For every set of channels C, every mapping $x: C \to M^{\aleph}$ provides a complete communication history. Note that $(C \to M^*)^\infty$ and $C \to (M^*)^\infty$ 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; we assume $\surd \notin M$).

We denote the set of valuations of the channels C by sequences

$$C \to M^*$$

by

$$\vec{C}^*$$

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

$$C \to M^{\aleph}$$

by

$$\vec{C}$$

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

$$x{\downarrow}i$$

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

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

we 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 only 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 we can find out in which time interval a certain message arrives, while in $\overline{x}$ we see only the messages in their order of communication without any indication of their timing.

We use both notations $x{\downarrow}$ and $\overline{x}$ as well as the concatenation introduced for streams x also for tuples and sets of timed streams by applying them pointwise.

Fig. 4 describes the syntactic interface of a component with the input channels x_1, ..., x_n of the sorts M_1, ..., M_n and the input channels y_1, ..., y_m of sorts M'_1, …, M'_m. In the theoretical treatment we assume for simplicity always the same sort M.

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

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

This function yields the set of output histories F.x for each input history x. Given the input history x a component with the behaviour F produces one of the output histories in F.x. We write F.x for F(x) to save brackets.

Only if a set-valued function on streams fulfils certain properties we accept it as a representation of a behaviour of a component. To give a precise definition of these requested properties we introduce a number of notions for set-valued functions on streams. A function

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

is called

- *timed*, if for all $i \in \mathbb{N}$ we have

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

  Then the output in the time interval i only depends on the input received till the i´th time interval. In the literature then F is called a *causal* function, too.

- *time guarded*, if for all $i \in \mathbb{N}$ we have

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

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

- *realizable*, if there exists a time guarded function[3] $f: \vec{I} \rightarrow \vec{O}$, such that for all input histories x we have:

  $f.x \in F.x$

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

- *fully realizable* , if for all input histories x we have:

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

We assume in the following that stream processing functions that represent the behaviour of components are always time guarded and fully realizable. For the set of functions

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

that are time guarded and fully realizable we write

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 we denote the set of all behaviours with arbitrary syntactic interfaces.

4.2 The Logical Specification of System Components

We describe components by set-valued functions on communication histories. Set-valued functions are isomorphic to relations. A set-valued function or a relation on streams can be described by logical means with the help of logical formulas describing the relationship between the input and output streams.

Example: A simple Store
We follow always the same scheme to describe a component. We describe the syntactic interface of the component Store by the data flow node given in Fig. 5. A store consists of a set of memory locations. For each memory location a value is stored. The values can be read and updated. Both reading or updating a value can be successfull or it can fail.

---

[3] 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 extends to functions.

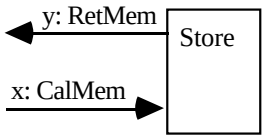

Fig. 5 Data Flow Node Giving the Syntactic Interface of the Store

Let MemLocs be the sort of memory locations and MemVals the sort of values that can be stored in the memory. The data model consists of the sorts of messages that are declared as in section 3.4 by Tab. 1. A straightforward formalisation of the behaviour reads as follows:

$$
\begin{aligned}
y \in \text{Store.x} \Leftrightarrow \quad & \#\bar{x} = \#\bar{y} \\
& \wedge \forall\, i \in \text{MemLocs},\, d \in \text{MemVals},\, k \in \mathbb{N}: \\
& \qquad \bar{x}.k = p \Rightarrow \bar{y}.k \in \{\text{ret}(p, \text{MemFail}), \text{ret}(p, \text{Ack})\} \\
& \qquad \wedge \quad \bar{x}.k = g \Rightarrow \bar{y}.k \in \{\text{ret}(g, \text{MemFail}), \text{ret}(g, v)\} \\
& \textbf{where} \quad p = \text{put}(i, d) \\
& \qquad\qquad g = \text{get}(i) \\
& \qquad\qquad v = \text{last}(\bar{y}\text{: } (k\text{-}1), i)
\end{aligned}
$$

Here v denotes the last successfully written value for the memory location i and

$$\bar{x}.k$$

denotes the k´th element in the stream $\bar{x}$. $\#\bar{x}$ denotes the number of elements in $\bar{x}$. $\bar{x}$:k denotes the prefix of the stream $\bar{x}$ of length k. The function last is specified as follows (let init_val be a given element of sort MemVal):

$$
\begin{aligned}
& \text{last}(\langle\rangle, i) = \text{init\_val}, \\
& \text{last}(z^\frown\langle\text{ret}(\text{get}(j), w)\rangle, i) = \text{last}(z, i) \\
& \text{last}(z^\frown\langle\text{ret}(\text{put}(j, d), w)\rangle, i) = \quad \textbf{if } i = j \wedge w = \text{Ack} \ \textbf{then } d \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \text{last}(z, i) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{fi}
\end{aligned}
$$

last(z, i) returns the last value written for location i in the result stream z. The logical specification of components is a very puristic description of the behaviour of components. A more pragmatic specification of component Store is given in the following section by state transition diagrams.

5. State Transition Systems

In this section we introduce a mathematical concept of a state transition system and techniques for describing it. We suggest state transition diagrams (STDs) for the description of the behaviour of components.

5.1 State Transition Models

State machines are a well-known concept of a system model. They are used both in practice and in theory. We work here with a specific version of state machines that corresponds well to our notion of a system component. By a state machine some internal detail of the system in terms of the states is provided. Therefore we say that state machines provide a *glass box view* of a system component. In the following section we provide another type of a glass box view of components described by the internal structure of a component.

A *communication pattern* for a set of channels C is given by a mapping $p \in C^*$ which assigns a finite sequence of messages to every channel in C. A mathematical state based system model uses state transitions. It includes in addition to the sets of the syntactic interface above a set

State

which denotes the set of states (the state space) of the component. We 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 produces a successor state and a communication pattern for its output streams. For this kind of state transition machines, we represent the set of possible updated states and outputs of a transition with the help of a predicate. Of course, sets of pairs of states and output patterns can be used here directly. 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 $x \in I^*$ representing the sequence of input message x(c) received in a time interval on the channel $c \in I$ of the component in state $\sigma \in$ State, every pair (σ', y) in the set $\delta(\sigma, x)$ represents a possible successor state and the sequence of output messages y(c) produced on channel $c \in O$ in the next time interval.

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

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

If the state transition relation contains cycles then the definition of F_σ is recursive. In this case, we cannot be sure that by the equation above the behaviour F_σ is uniquely specified. Therefore we define F_σ by the largest (in the sense of pointwise set inclusion) time guarded function that fulfils this equation.

The first (existentially quantified) part of the left-hand side of this defining equation handles the case where at least one of the input patterns applies. The second part treats the case where for the input stream x none of the input patterns applies.

This case is mapped onto arbitrary output called *chaos*[4] . This definition is justified by the principle that, if nothing is specified for an input pattern, the system may react by arbitrary output. This definition, moreover, guarantees that the function F_σ is always fully realizable.

If we do not want to associate a chaotic, but a more specific behaviour with input situations where no input pattern applies, we can work with default transitions (for instance time ticks) or simply drop the second clause in the definition. Working with chaos, however, has the advantage that adding input patterns for input for which no pattern applied so far is a correct refinement step in the development process.

Our model of the behaviour of a component works with timed input and output streams. Since the input and output patterns as introduced above do not refer to the timing of the messages, in the definition we work with time abstractions of the input and output streams. Of course, we may also work with input patterns that refer to time. The explicit time concept in the behaviour model of a component allows us also to deal with priorities in transitions.

We even permit transitions with empty input patterns called spontaneous transitions. If we work with our assumption that for a transition on every channel there is at least one time tick, then a spontaneous transition is only enabled if there are no messages but only time ticks on the input channels.

Note that we can give along these lines a precise treatment for sophisticated concepts like priorities and spontaneous transitions due to our carefully chosen semantic model that includes time. 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.

5.2 Description of State Transition Systems

We describe state transition systems by state transition diagrams. To demonstrate the use of state transition diagrams (STDs) for the description of the behaviour of components we give a first simple example.

Example: A simple store
The state space of the state machine contains only one attribute declared by

a: MemLocs → Data

Initially all memory cells have the initial value init_val which is a distinguished element of MemVals

Initial: a(i) = init_val

In the case of the store we work only with one node in the state transition diagram. Then, we might better not use a state transition diagram at all, but work with a set of transitions very similar to TLA (see [Abadi, Lamport 90]). □

We do not give a formal translation of STDs to state transition systems. It is rather technical, but quite straightforward.

[4] If no input pattern applies we assume that a specific behaviour is not required.

5.3 Description of the Behaviour of System Components

We have shown that state transition systems can be used to describe the behaviour of system components. A state transition system can be described by a state transition diagram or equivalently by a state transition table. Such descriptions receive much more acceptance in practice than purely logical representations of state transition systems. However, without much overhead we may describe state transition systems by tables or diagrams and translate these into logic.

x:put(i, d) / y:ret(put(i, d), Ack) {a'(i) = d ∧ ∀ j, j ≠ i: a'(j) = a(j)}
x:put(i, d) / y: ret(put(i, d), MemFail) {a' = a}
x:get(i) / y:ret(get(i), a(i)) {a' = a}
x:get(i) / y:ret(get(i), MemFail) {a' = a}

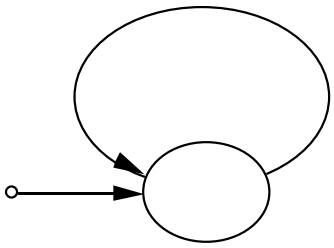

Fig. 6 State Transition Diagram

The transition rules can be gathered into a table. This is shown for our example of Fig. 6 by Tab. 2. This way we avoid long formulas in the state transition diagrams.

Tab. 2 Transition Table

name	input	output	postcondition
Write	x:put(i, d)	y:ret(put(i, d), Ack)	a'(i) = d ∧ ∀ j, j ≠ i : a'(j) = a(j)
WriteFail	x:put(i, d)	y: ret(put(i, d), MemFail)	a' = a
Read	x:get(i)	y:ret(get(i), a(i))	a' = a
ReadFail	x:get(i)	y:ret(get(i), MemFail)	a' = a

We can use the table in connection with a state transition diagram as given by Fig. 7 which only refers to labels included in the table.

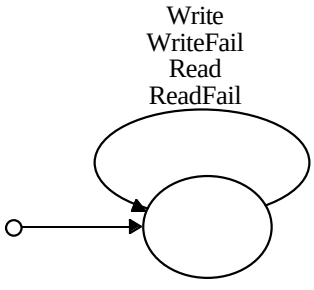

Fig. 7 State Transition Diagram with Abbreviations for Transitions

The state transition diagram of Fig. 7 contains only one node. So, as pointed out, it is an overkill to work with a state transition diagram at all. However, we may replace the state transition diagram by one that contains two nodes. We may also work with a state transition diagram where each transition is split into two, one that accepts the input and the next one which produces the output. Then we need a more detailed state space to store the location (and the value) for which reading or writing is required. So the state space is given by the declaration of the attributes

a: MemLocs $\rightarrow$ Data, c: CalMem

The table Tab. 3 lists all the transitions and Fig. 8 gives the state transition diagram.

Tab. 3 Transition Table for the Decoupled State Transition Diagram

name	condition	input	output	postcondition
Com		x:e	-	c' = e $\wedge$ a' = a
Write	c = put(i, d)	-	y:ret(put(i, d), Ack)	a'(i) = d $\wedge$ $\forall$ j, j $\neq$ i : a'(j) = a(j)
WriteFail	c = put(i, d)	-	y: ret(put(i, d), MemFail)	a' = a
Read	c = get(i)	-	y:ret(get(i), a(i))	a' = a
ReadFail	c = get(i)	-	y:ret(get(i), MemFail)	a' = a

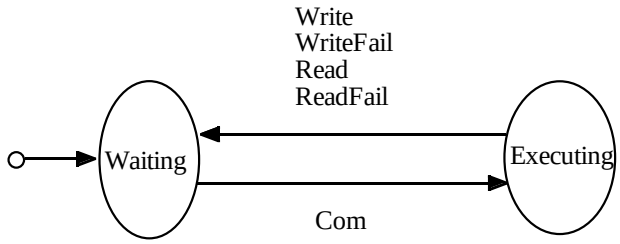

Fig. 8 State Transition Diagram for the Store with Transition Rules Separated into Input and Output

This refined version of the store has the same behaviour as the store described by the diagram above. □

The example above shows a technique for detailing a state transition diagram by splitting its nodes and transition rules. If all nodes have only transition rules where either the input pattern or the output pattern is empty the state transition diagram is called *decoupled*. They correspond closely to the I/O-machines of [Lynch, Stark 89] where each state transition is labeled by exactly one input or output action.

So far, we have shown examples only for the description of components. In the next section we show how to put together components into a network such that they cooperate. Since state transition diagrams describe data flow components, for which a composition by data flow nets is well-understood, we can use this concept of composition immediately for state transition descriptions. We demonstrate this by a simple driver component that cooperates with the memory.

Example: Store Driver
A driver is a component that controls the access to the memory for many processes. It receives calls and forwards them to the memory. It may try again if a memory call fails. However, it may stop trying at any time, even if before it tried at all. We use our general specification scheme again.

The syntactic interface of the component Driver is described by a data flow node as it is given in Fig. 9.

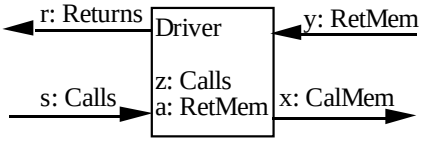

Fig. 9 Data Flow Node Giving the Syntactic Interface and the State Attributes of the Driver

The state space of the Driver with its attributes is described as follows:

z: Calls, a: RetMem

The state transition diagram given in Fig. 10 specifies the behaviour of the component Driver.

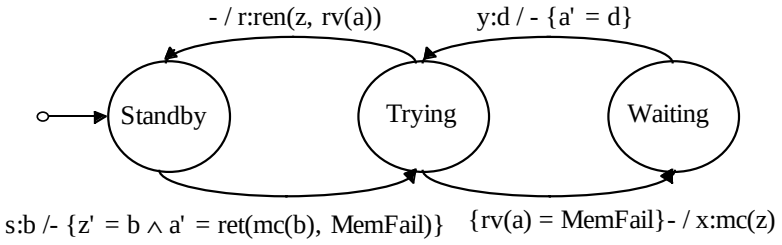

Fig. 10 State Transition Diagram for the Driver

The driver is designed to cooperate with the store. The driver may try forever in the case of memory failures, if we do not assume any fairness assumptions. To get rid of this "unfair" behaviour, we may add the liveness condition

$$\# \bar{r} = \# \bar{s}$$

to the component specification that expresses that every input leads to an output under the assumption

$$\# \bar{x} = \# \bar{y}$$

that expresses that all calls transmitted on x get eventually served. It is very helpful that the state transition description can be combined with logical equations to express liveness properties. □

The technique of state transition diagrams gets more flexible by working with sets of such parameterized diagrams.

Example: RPC
Our driver in the example above can handle only one request at a time. To be able to handle more than one request at a time (more precisely one request for each processor) we introduce a set PrIds of processors and define the state space as follows

$$v : PrIds \rightarrow state(z: Calls, a: RetMem)$$

We work with a multithreaded diagram (or more precisely with a family of single threaded diagrams). For all process identifiers $p \in PrIds$ we give one transition diagram.

We use the following straightforward abbreviation. For a record r of sort

cons(sel_1: M_1, ..., sel_n: M_n)

we use the convention that

with r: E

stands for

E[sel_1(r)/sel_1, ..., sel_n(r)/sel_n]

Fig. 11 shows the state transition diagram of the multithreaded driver. Initially, all default outputs are memory failures:

Initial: rv(a(v(p))) = MemFail

Note that the syntactic interface of the multithreaded Driver coincides with that of the single-threaded driver. □

In a multithreaded state transition diagram a set of nodes is active. Each transition moves the control of one of these nodes to another one. Chaotic default transitions are only allowed if for none of the active nodes a transition is enabled.

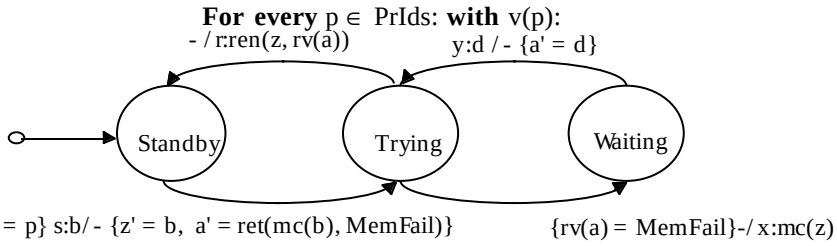

Fig. 11 State Transition Diagram for the Multithreaded Driver

Note: If we use a set of nonconnected state transition diagrams which work with disjoint sets of channels we can decompose the data flow node in a set of data flow nodes that work in parallel. Vice versa, a data flow diagram where we have a transition diagram for each node is a multithreaded state transition system. □

More research is needed, however, to work out concepts that allow us to encode more powerful logical descriptions into tables and diagrams in a readable way (see [Parnas, Madley 91]).

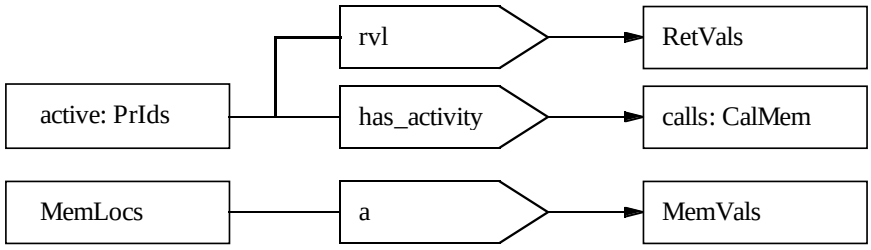

Fig. 12 Data Model for the State of the Multituser Memory

For many applications it is appropriate to distinguish between regular and exceptional system activities. What is called regular and what is called exceptional has to be

decided by application considerations. Often it is helpful to separate regular cases from exceptional cases when describing system behaviours.

For the memory only one reader or writer can be active in the memory at a time. A memory where several readers/writers are active simultaneously is specified with the help of a more complex data model. It is given in Fig. 12.
Every member in active is related to a call by the function has-activity.

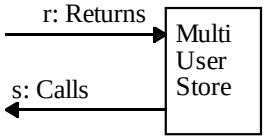

Fig. 13 Data Flow Node Giving the Syntactic Interface of the Store

Now we can describe the behaviour of the multiuser memory by the state transition diagram. Fig. 13 gives this transition diagram. The transition rules are gathered in Tab. 4.

Tab. 4 Transition Table for Fig. 14

name	precondition	input	output	postcondition
Write	has_activity.p = put(i, d)	-	-	rvl(p) := Ack, a(i) := d
WriteFail	has_activity.p = put(i, d)	-	-	rvl(p) := MemFail
Read	has_activity.p = get(i)	-	-	rvl(p) := a(i)
ReadFail	has_activity.p = get(i)	-	-	rvl(p) := MemFail
ComIn	$\neg p \in$ active	r:ca(p, c)	-	has_activity(p) := c, rvl(p) := MemFail
ComOut	$p \in$ active $\wedge$ has_activity.p = c	-	s:ren(c, val(p))	$\neg p \in$ active'

We can use the table in connection with a state transition diagram as given by Fig. 6 which only refers to labels included in the table.

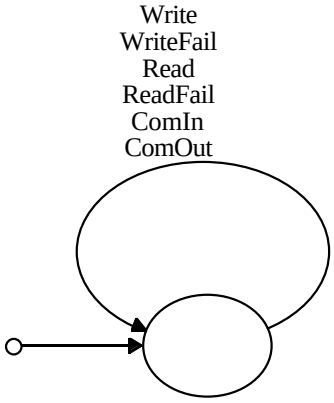

Fig. 14 State Transition Diagram for the Multi-User Store with Abbreviations for Transitions

Again we may addlivesness conditions such as fairness requirements that make sure that every call eventually returns. If we assume that the store is fair and returns a positive acknowledgement eventually if the Driver tries long enough. Another

possibility is to work without such a fairness assumption and let the driver retry until it receives a positive acknowledgement.

6. Distributed Systems and Their Description

In this section we introduce a mathematical notion of distributed systems and techniques for their description. Our notion is based on a data flow model of distributed systems.

6.1 Mathematical Models 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 we speak of a *composed distributed system* and in the later case we speak of a *nondistributed system*. Fig. 12 shows the data flow representation of a simple example of a distributed system.

We 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 syntactic interface (I, O) is described by the mapping

$$\nu: N \rightarrow Ç$$

that associates with every node a component representing a behaviour (an interface behaviour). As a well-formedness condition we require that for all component identifiers $i, j \in N$ (with $i \neq j$) the sets of output channels of the components $\nu(i)$ and $\nu(j)$ are disjoint:

$$Out(\nu(i)) \cap Out(\nu(j)) = Ø$$

In other words there are no name clashes for the output channels and thus, each channel has a unique source. We denote the *set of nodes* of the net by

$$Nodes((\nu, O)) = N$$

We denote the *set of channels* of the net by $Chan((\nu, 0))$. It is defined as follows:

$$Chan((\nu, O)) = O \cup \{In(\nu(i)): i \in N\} \cup \{Out(\nu(i)): i \in N\}$$

The set I of *input channels* of the net is defined as follows:

$$I = \{In(\nu(i)): i \in N\} \backslash \{Out(\nu(i)): i \in N\}$$

The channels in the set

$$\{Out(\nu(i)): i \in N\} \backslash O$$

are called *internal*. By these definitions, every channel is either internal, an output channel, or an input channel.

According to this definition a distributed system has like all other description techniques a static (syntactic) part and a dynamic (semantic) part. The set of component identifiers, the channels and the way how they connect components form the static part of the distributed system while the function ν, which assigns behaviours to the components represents the semantic part of a distributed system.

In recent years the theoretical and practical interest in distributed *dynamic* systems increased. Dynamic systems are also called systems with *mobile communication* or *mobile systems* for short. In a dynamic system the set of components and channels changes over its lifetime. Such systems can also be described by the type of system models introduced above. However, more refined models are needed where either the net is used as a state of the system that changes over time or possibly infinite nets are considered that comprise all components that may be created over the lifetime of the system. In the case of dynamic channel creation similar ideas can be used (see [Grosu 94], [Broy 95a], [Grosu et al. 95]).

6.2 Black Box Views onto Distributed Systems

Given a distributed system in the form of a data flow net with the set I of input channels we get an abstraction of it (which we call its black box view) by mapping the distributed system onto a component behaviour $F \in Ç[I, O]$.

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 $g: D \to R$ and a set $T \subseteq D$ we denote by $g|_T: T \to R$ 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[5)] for all equations on the streams induced by the channel of the network.

6.3 The Description of Distributed Systems

A distributed system, as we defined it in the previous section, can be described graphically by a data flow network (called CD for communication diagram in GRAPES, see [Grapes 90], called block diagram in SDL, see [SDL 88]). A data flow network is a labeled directed graph. This way the structure of distributed systems can be represented.

Data flow nets are illustrative diagrams (see Fig. 12) that give a structural view onto information processing systems. Their weak points are that they only describe the static, syntactic properties and do not indicate any semantic properties. They are only helpful as long as the set of the components of a system is static. As soon as the sets of components get dynamic, it is impossible to provide a static structural view of systems by data flow nets. Nevertheless, even for object oriented techniques that include the dynamic creation and deletion of objects, data flow models can be useful.

[5)] According to the fact that we only consider delayed timed behaviours it can be shown that in the deterministic case for each input there is always a unique fixpoint. In the nondeterministic case, there may be several fixpoints, of course, each of which corresponds to a computation of the net. Therefore a sophisticated logic of least fixponts is not needed.

We can represent a complete class of objects in object oriented programming by a single data flow node. Each such node represents the set of objects of that class operating in parallel. It is not very helpful, of course, to represent the structure of a class itself by a data flow diagram, in general.

A data flow net graphically represents the components of a system and their communication interconnections by the channels. Thus a data flow net provides a static view on a set of components and their connections. For systems with a more dynamic structure as mentioned above, where the set of components as well as their interconnection by communication links do change over the lifetime of a system, a classical data flow net can only provide a snapshot. However, we can also use (even infinite) data flow models that incorporate all components (or classes of components) that may exist over the lifetime of the system. Then each component goes through three phases. In the beginning it is inactive. Then it gets active (in the slang of object orientation „it gets created“). After a while it may become inactive again.

The exact meaning of data flow diagrams is not always described precisely for the software engineering methods used in practice, although most of these methods advocate versions of data flow diagrams as part of their description techniques. Nevertheless, as well known, we can formalise the meaning of data flow nets by stream processing functions (see also [Broy 95b]).

As mentioned above, the weak side of data flow nets are their limited possibilities to describe the behaviour of systems. They indicate which components exist and may exchange messages, however, this does not say much about the causal relationships between the exchanged messages.

Of course, we can annotate data flow diagrams with information about the sorts of the messages exchanged on a data link. This provides a description of the syntactic interfaces of the components of a system.

The driver can be combined with the component Store. This composition is asynchronous as it is defined in asynchronous data flow. By combining the driver and the store, we obtain a component which again can be described by a state transition machine. Its state space consists of the product of all the state spaces of the subcomponents. In addition, we need buffers to store the messages sent on internal channels until they are received.

Such buffers can be avoided, of course, if we work with decoupled state transition diagrams defining state machines that do all their transitions with input from internal channels in two steps. In the first step the input is received and in a successive step without input the output is produced. By such a refinement of state machines into decoupled state machines the buffers have to become part of the state space.

Example: Composing the Driver and the Store into a Data Flow Net
The interface of the driver and the store fit together such that we may compose them as shown in Fig. 15. In cases where the channel identifiers do not match we have to rename them.

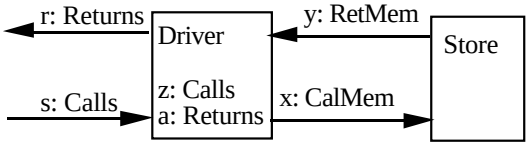

Fig. 15 Composition of the Driver and the Store

This composition can also be described by a textual syntax. As the diagram suggests, the channels x and y are hidden and not part of the interface of the black box view.

The driver and the second version of the memory component of section 5.3 are already in the form that allows us to work without buffers when combining their state spaces into a state space for the state transition description of the composed system. □

Of course, the buffer can only be completely avoided, if the decoupling is broken down onto the level of singular steps and in every node every input can be accepted. Only then no additional buffers are needed to store messages on input channels provided the system runs fast enough to process all inputs immediately.

6.4 State Views onto Distributed Systems

If all components of a distributed system are described by state machines we easily may obtain a state view of a distributed system. To keep our presentation simple we assume that for all components their sets of attributes are pairwise distinct and that their state transition description is decoupled and in every state every input can be immediately processed. Then we obtain a state view of a distributed system simply by joining the state views of all its components. The state transitions are triggered by the state transitions of the components and the exchange of messages. Independent machines can carry out transitions in parallel.

7. Processes

In this section we introduce the mathematical concept of a process and in addition techniques for describing processes. Processes are a concept describing the internal communication behaviour of distributed systems.

7.1 Mathematical Models of Processes

The notion of a process is crucial in software engineering. This can be seen by the term information *processing*. The notion of a process is very general and used in many different ways in computer science. We understand a (discrete) process as a model of the flow of causally related activities in a distributed interactive system. It is represented by a mathematical model consisting of the actions and their causal relationships. A process is a family of actions that are in some causal relationship. In the case of interacting systems, an action is triggered by a number of messages that have been received. When it is carried out, it results in sending a number of messages that may cause further actions. If in a process an action a1 is directly causal for an action a2 there must exist a message sent from action a1 to action a2.

Each instance of sending or receiving a message is called an *event*. Each event is caused by the sender of the respective message. 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 as messages that are sent by a timer.

Pioneering work on the notion of a process was done by Carl Adam Petri who suggested by the so-called Petri nets a fundamental technique for describing concurrent processes. Strictly speaking Petri nets are a kind of automata that describe concurrent processes. An explicit description of processes is obtained by the so-called *occurrence nets* (see [Reisig 86]) which are Petri nets without cycles and conflicts.

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

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

that associates with all of the arcs of the net exactly one message. Then an action is represented by a behaviour relation that relates the messages in its input lines to the messages in its output lines.

As a result of our definition, a process is a special case of a data flow net. In contrast to general data flow diagrams, however, through each of its channels exactly one message is sent. So arcs correspond in process nets to communication events (message transmission) and nodes correspond to action events (receiving and sending of messages).

There are many variations of mathematical models for processes including trace sets and action structures (see [Broy 91]). We do not introduce a more explicit mathematical model of a process here but consider processes as a special form of distributed systems. The relationship of our notion of process to data flow nets is explained in detail in section 9.4.

7.2 The Description of Processes

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 as a spin off of SDL (see [SDL 88]) called *message sequence charts* (see [MSC 96]) or in objectory (see [Booch 91]) called *use cases*.

A sequence chart defines a trace of communication events for the system and for its components that take part in the interaction. Fig. 16 provides an example of a sequence chart. It provides the same information as the process diagram given by Fig. 17.

Processes of interactive systems can be 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 a 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.

Graphically each event can be represented by an arc and by the two involved actions (components) (sender and receiver). Since causality between events is always realised through the actions (components), an event e1 can only be directly causal (triggering) for an event e2 of another component, if the sender of e2 is the receiver of e1.

This way we get component-based process views. Our kind of processes are obtained as specific subprocesses of the behaviours of structural system views. Of course, in a structural view several processes and several tasks may be carried out in an interleaved manner.

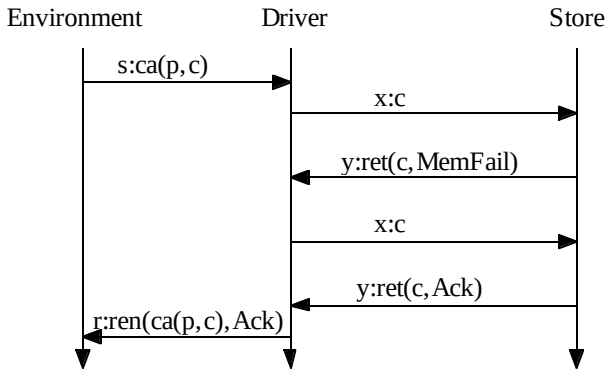

Fig. 16 Sequence Chart of a Case of Interaction between the Driver and the Store for c = put(...)

In the processes we do not always explicitly include the data flow between the different activities of one component. This may, however, be necessary if the process view is to be refined into a structural system view during the system development. Similarly, the access to a common data base may not be modelled explicitly in a first development step. However, later such a modelling is necessary to obtain a correct representation of the data dependencies between the involved actions. Fig. 17 gives a process view of a standard process for our example application.

The behaviour of a component connecting a number of input events with a number of output events occuring in a process view can be described again by a data flow network or by a state machine (called PD in [Grapes 90]). A formal definition of the interpretation of processes as runs of systems described by data flow graphs is given in section 8.4.

8. Modelling Techniques

Description formalisms give - similar to programming languages for describing algorithms - a solid basis for representing models. There is as much freedom in how to use a description formalism as there is freedom in using a programming language to provide an algorithm for a problem.

For the development of information processing systems we have to develop an adequate model of the information processing task. Since it is difficult or even impossible to describe a complex systems in full detail in one document, we structure the description by providing models and abstractions. An abstraction is a simplified description of the system. A useful abstraction allows us to concentrate on significant aspects of the system reducing the overall complexity of the modelling task.

A *view* shows us the system under an individual perspective highlighting specific aspects. So a view is always an abstraction. However, since we generally work with a fixed collection of views, each of the views provides a uniform type of abstraction. Certain views are well accepted in software engineering. For instance, data views, which provide a view on the data aspects of an information processing system, are

well received and can be found in terms of E/R-models in most development methods. Process modelling techniques are used in many methods, too. Recently, they have gained more attention. It is widely recognised by now that process views can help significantly in understanding an application problem and properties of solutions, especially in the communication between the software engineer, the application expert, and the client.

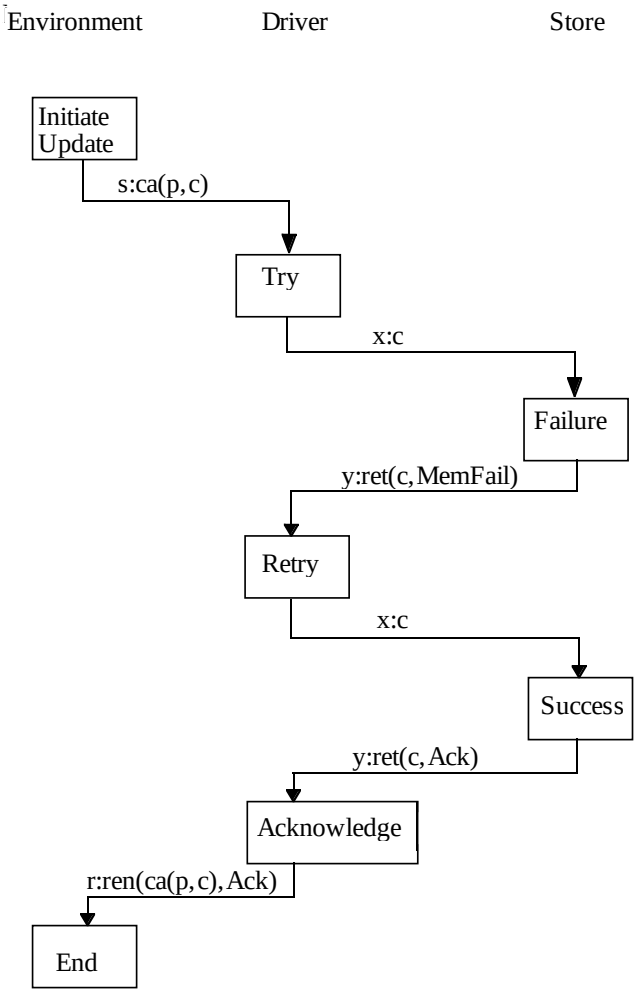

Fig. 17 Process Description of a Case of Interaction for a Call c = put(...)

Nevertheless, how to make a systematic use of process modelling techniques in the software and system development is still not fully understood. A process represents the history of the run of a system in terms of the actions performed by a system and their causal relationship. Since often the processes of a system are too large to show them in full details the behaviour of a system is illustrated by sample subprocesses (called use cases) handling individual cases. There is a general agreement that besides data views and structural views this modelling of processes is essential for the development of interactive systems.

Process modelling techniques can be used in many application areas. In telecommunication, for instance, message sequence charts are advocated (see [MSC 96]). Object oriented techniques provide *use cases*, too. We agree that process modelling is a helpful concept in most application areas of information processing. We believe, too, that a systematic use of process modelling techniques can be helpful, in principle, both in technical and in business applications.

In the early phases of system development often called system analysis or requirement engineering, we work out the following five complementary system views:

- data view (also called information model),
- process view (dynamic view),
- structural view (also called organisational or architectural model),
- behaviour view (interface model, behaviour history model, black box view),
- state view (state transition model).

Software engineering has provided many different description techniques for all these modelling aspects (see above). What is lacking in many methods is the well-worked out semantical and methodological integration of these views (see below). In particular, it is not always clear what the role of process views is in system and software engineering from a methodological point of view.

In the following, we sketch the role of the data view, the behaviour view, and the process view in the course of development of a system. Structuring descriptions is one of the most important goals when we try to keep the description of large systems comprehensible. This is also supported by levels of abstractions. Ideas, how the levels of abstractions are connected by refinement notions, are shown in section 9 on the development method.

8.1 Building the Data View and the State View

We suggest to use data views mainly for modelling static aspects of the data relevant for a system. There are strictly object-oriented approaches that suggest to mix static and dynamic aspects of a system into one E/R-diagram. They suggest also to model each business case by an object. This object includes all the attributes needed to represent the state of the individual business case and all the methods needed to change the business case attributes. This is suggestive as long as such an object is modelled as being passive. It becomes questionable as soon as such an object gets active and calls methods of other objects. This contradicts the principle of object orientation that has the goal to represent reality as "naturally" as possible.

We can also give a data view of the state of a distributed system (ν, O) by an E/R-diagram that defines a number of components in terms of the set of components N and provides a state space for each of these components by a number of attributes with their sorts.

8.2 Process Modelling

In this section we discuss shortly how to use process modelling techniques in system development. We model actions in a process by a data flow node that formally represents a component. However, a component representing an action has an

especially simple structure. On each of its input arcs it receives exactly one message and on each of its output arcs it issues exactly one message.

Note that we do not require a strict behaviour where all input messages have to be available before all the output messages are generated. Some output messages may require only a subset of the input messages. In a complete modelling of an action all the information an action needs is provided by its input messages. Since an action is represented formally by a data flow component we can describe it by the same techniques that we use to describe components.

There is a rich body of research on the topic (Petri nets, process algebras like CCS and CSP, event structures, etc.) how to describe systems by sets of processes. However, most of this work can only be used as a theoretical foundation and does not help much in concrete description tasks. So more work needs to be done to make these theoretical approaches practically useful.

8.3 Structural System Views

In a structural system view the structural decomposition of a system into a family of components is shown. A complete description of a structural system view by a data flow net where for each of its data flow nodes black box behaviours are given contains enough information to simulate system behaviours, provided the behaviour of all components is described by means for which an operational interpretation is given. Structural system views are easily provided by data flow diagrams as shown in Fig. 12.

In a system development there may be several structural views, for instance, indicating the logical structure and the physical structure of a system. Building a structural view for a component for which we only have a black box view is a difficult task. This is why the importance of architectures is so much recognized, today. We have to decide upon the set of subcomponents, how they are connected and how they interact. For this process views may help. Finally we have to give the black box views of the subcomponents.

8.4 Behaviour Views of Components

A structural system view on a system providse a description of the overall system architecture. It also provides a description of the system behaviour if for each component a behaviour is fixed. We distinguish between a black box view and a glass box view of a component. In a black box view the behaviour of a component of an interactive system can be represented either by a trace set or by a stream processing function. In a glass box view the behaviour can either be represented by a state machine or by an interactive distributed system.

Often, it is advisable, first to take into consideration only the regular aspects of the behaviour of components. We call this the behaviour in nonexceptional cases. Only in a second step we consider the exceptional cases. In other words, the behaviour is refined to nonexceptional cases in a second step. We distinguish the following classes of exceptional cases:

- external input events that are expected, but do not occur (within certain time bounds),

- external input events that occur unexpectedly,
- external output events are expected that cannot be generated by the system because of lack of information.

The same distinctions can be made for the internal events of distributed systems. In any case, it is advisable for complex components to start with a simplified version and then to refine it to a more sophisticated one.

9. Development Method

A development method gives hints and suggestions, how to use the description and modelling techniques in a systematic way to develop a computer-based information processing system. In software engineering, the development is carried out by going through several development phases. We speak of a *development process model* when these phases and the documents are fixed.

9.1 Development Process Model

In software engineering the discussion of the development process model plays a central role. This has clear reasons. The development process model governs the whole development process and is therefore decisive for the structure, the economy, and the quality of the development.

Often religious fights are fought in software engineering whether the waterfall model, the spiral model, the experimental or the evolutionary prototyping model, or object oriented development methods provide the best development process models. Not enough practical data are available and not enough theoretical work has been invested to analyse, evaluate, discuss, characterise, relate, and formally specify these different development models.

Mathematical techniques can help to relate 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.

A more rigorous formalization of the development process is suggestive. Development processes are instances of processes as they are discussed in theoretical computer science. So we can apply all our notions from process theory here, too. In the discussions about development process models, it is often not recognised that we have to distinguish between the overall structure of the development documents and how they are related on one hand and the temporal order in which they are produced on the other hand. So we can talk about the statics (the logical dependencies between the documents) and the dynamics (the order in which the documents are produced) of the development method and the involved activities.

9.2 Refinement

In software engineering, systems are described by a number of complementary views and on different levels of abstractions. 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. View integration is dealt with in section 9.4. In this section we treat refinement.

Much theoretical work has been devoted towards the refinement notion. After the more pragmatic and informal ideas of stepwise refinement developed by Dahl, Wirth and many others in the seventies, much formalization has taken place converging essentially into the following two concepts:

- *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). This allows us also to replace component specifications by logically equivalent component implementations.
- *representation refinement* (a special case is *data refinement*): In representation refinement we 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.

9.2.1 Data Refinement

For data structure descriptions, property refinement is quite simple. Given a signature (S, F) and a set of algebras Alg, by property refinement we add further requirements to our specification. This way, the set of models Alg is reduced to a subset $Alg' \subseteq Alg$.

More sophisticated is the idea of the refinement of the data representation. However, this is also well-understood by now. To refine the representation of a sort Asort we need a representation sort Rsort and functions

α: Rsort $\rightarrow$ Asort abstraction function

ρ: Asort $\rightarrow$ Rsort representation function

that relate the elements of these two sorts. The function ρ provides a representation for each element in Asort. In general, we work with relations ρ or with sets of representation functions to be able to allow many representations for one abstract data element. For our purpose it is sufficient to work with one instance of a representation function.

The function α maps the elements in Rsort that are used as representations onto their abstractions in Asort. The functions α and ρ have to fit together to define a proper idea of abstraction and representation. This is expressed by the following equation:

$$\alpha.\rho.x = x$$

Moreover for all functions on the sort Asort we have to provide representations. For instance for a function

$$f: Asort \rightarrow Asort$$

we have to find a function

$$f': Rsort \rightarrow Rsort$$

such that for all elements x in Asort we obtain

$$f.x = \alpha.f'.\rho.x$$

This requirement can be generalised to functions with more complex functionalities.

9.2.2 State Transition System Refinement

Since state spaces are only special cases of data models, the refinement concepts for data models carry over to state spaces immediately. As long as the sorts of the input and output channels are not refined we can simply work with property refinement. If not only the state space is to be refined but also the sorts of the messages and the number of input and output channels we should use the concepts of component refinement as introduced in the following section.

9.2.3 Component Refinement

There are with three fundamental concepts of the refinement of the interface view of system components

- property refinement,
- interface refinement,
- glass box refinement.

By property refinement we do not change the syntactic interface of a component but add properties to its specification. This reduces the set of possible outputs.

In interface refinement we may change the syntactic interface, but insist on a well-specified relationship between the behaviour of the refined component and the original one.

In glass box refinement we replace the black box view by a glass box view, which is provided either by a state machine or by a distributed system. The black box behaviour associated with the glass box view is requiired to be a property refinement of the original black box view.

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

$$\hat{f} : Ç[I, O]$$

is called a *refinement of a component*:

$$f: Ç[I, O]$$

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

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

We write then

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

Of course, this refinement notion can be carried over to state machines. For a systematic use of refinement concepts, *compositionality* (which is the mathematical version of *modularity*) of refinement is essential. Compositionality guarantees that, if we replace a component of a system by its refinement, we obtain a refinement of the overall system.

Property refinement does not change the syntactic interface of a component. 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. We introduce what is called an *upwards simulation* in the literature. We consider two components with different syntactic interfaces:

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

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

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

is called a *channel history refinement* if

$$R ; A = Id$$

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

$$A_I \in Ç[\hat{I}, I], \qquad R_I \in Ç[I, \hat{I}],$$
$$A_O \in Ç[\hat{O}, O], \qquad R_O \in Ç[O, \hat{O}],$$

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 $\hat{F}$.

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

9.2.4 Refinement of Distributed Systems

A distributed system is refined by refining its components. For each of its component we may use a property refinement. It is shown in [Broy 92] that a refinement of individual conponets of a network is guaranteed to lead to a property refinement of the overall distributed system. We may refine each component either

into a state transition machine or into a distributed system. Also this leads into the hierachical refinement of the distributed system.

Of course, we may also refine the communication histories for the internal channels of a distributed system by an interaction refinement. It is shown in [Broy 92] that this can be done in a modular way, too.

9.2.5 Process Refinement

Since in our approach, processes are only a special case of distributed systems the concepts of distributed system refinement can be used for process refinement, as well. We can apply all three concepts of refinement of data flow nets such as property refinement for actions, interface refinement and the glass box refinement for the refinement of processes. Glass box refinement allows us to replace an action by a process or to describe an action by a state machine.

A central notion of refinement is decomposition. We can use static or dynamic notions of decomposition of components, states, or processes.

9.3 The Modelling Universe

In the development of a system, we fix the above-mentioned five views and work out respective models for representing them:

- information (data),
- interface,
- subsystem behaviour by state change (state transitions),
- structure (organisation, distribution),
- process (business cases, use cases, message sequence charts).

These views have to provide a consistent integrated view of the systems.

For all applications the details to be described by the corresponding modelling notions have to be agreed on and to be worked out during the development process. This means that the five complementary views are refined through several levels of abstaction.

9.4 View Integration: Processes and the Structural System View

Given a complete system description, say by a structural system view (a hierarchical data flow graph), and a state transition description for each atomic component (a data flow node) the overall behaviour of a system is fixed. Then we can associate processes with such a system by unravelling the cycles of the data flow graph. This leads, in general, to infinite acyclic data flow graphs corresponding to a set of processes each representing a run of the system. In this unrolling of the data flow nodes the information flow between the actions of one component is made explicit.

A *process* is represented by an acyclic data flow net (ν_p, O_p) and a valuation function

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

that associates with each of its arcs exactly one message.

In the view integration we have to relate the process view to the structural view of a distributed system. This means that we have to define when a process is a consistent view of a data flow net. To formalize this motive, we define what it means to call a process a *run* of a data flow net (ν, O) with input history $x \in \vec{I}$. We require that there is a function

$$\kappa: \text{Nodes}((\nu_p, O_p)) \rightarrow \text{Nodes}((\nu, O))$$

that associates with every action of the process a data flow component. Intuitively κ(a) yields the component that models (carries out) the behaviour of the action a.

Every arc in the process corresponds to communiction event. Each of these events occurs for one of the channels of the data flow node. To model this correspondence, we require that there exists a function

$$\chi: \text{Chan}((\nu_p, O_p)) \rightarrow \text{Chan}((\nu, O))$$

that associates with every arc in the process a channel in the data flow net such that for every action represented by a node $i \in \text{Nodes}((\nu_p, O_p))$ each channel in its set of input channels is mapped onto an input channel of the node onto which the node i is mapped. The analogous condition is required for the output channel:

$$\{\chi(c): c \in \text{In}(i)\} \subseteq \text{In}(\kappa(i))$$

$$\{\chi(c): c \in \text{Out}(i)\} \subseteq \text{Out}(\kappa(i))$$

Since several channels (representing communication events) in the process are mapped onto the channel in the data flow network we assume that for each channel $c \in \text{Chan}((\nu, O))$ a linear order on the sets of channels (arcs)

$$\{c': \chi(c') = c\}$$

in the process is defined. This order is used to determine in which order the messages in the process are sent on the channels on the system. Of course, this order has to be consistent with the causality order in the process. This way we can associate a finite stream υ(c) of message s with each channel $c \in \text{Chan}((\nu, O))$ of the data flow net. It consists of the messages in the process associated with that channel.

The process (ν_p, O_p) is called a *run* of the system (ν, O) for input x, if there exists a valuation function

$$y: \text{Chan}((\nu, O)) \rightarrow M^{\varpi}$$

such that

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

and

$$\upsilon(c)|_{\text{In}(\nu(i))} \sqsubseteq \overline{y(c)}$$

In words, the process provides communication histories that form a prefix of a computation of the net.

In many cases, processes are not used to model initial segments of system computations. Then a process may consist of all messages of a particular subset M' of the set of messages M. Then we require

$$\upsilon(c)|_{\text{In}(\nu(i))} \sqsubseteq M'\copyright \overline{y(c)}$$

where M'©y denotes the family of streams obtained from y by filtering out all messages in the set M'. This allows us to model transactions by processes.

9.5 The Role of Process Views in the Development

In the development, process views are used as a means for documentation and analysis for the following purposes:

- In the early development phases instances of processes and process views can help in the requirement capture. They are helpful to illustrate representative use cases. They can be a valuable means for the communication between an application expert and the system analysis expert.
- As soon as a complete structural system view is provided, process views can be used to illustrate the system behaviour by simulations. One may also think of a model checking tool that checks whether only the required processes are possible behaviours of the structural system view.

We can associate a set of states with every process. Formally, the set of states can be represented by the set of prefixes that, in fact, represent all partial executions (snapshots) of a run of the system. Of course, such a representation of the states of a process is much too abstract and therefore not very helpful. However, by introducing appropriate names and representations for the states we can gain additional views on a process and relate them to the state transition view.

As we have seen, a component can be described by a state machine. The state machine can be systematically derived from the process view. For every component we can construct a local view onto the process and represent this by a set of states. It can be understood as the projection of the state view of a process onto a component. Then we can describe the behaviour of the component concerning the process by a state machine.

9.6 View Integration by Axiomatic Specifications

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 which are given by 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 in [Hußmann 95]. There a widely used practical software development method is defined by translating it into axiomatic specifications.

10. Conclusions

To work out a pragmatic, practically usable method for the development of large software systems that is properly founded on a scientific, mathematical basis is one of the challenges for computer scientist and software engineers. Such a method would allow to include in a flexible way formal methods for the specification,

refinement, and verification of system parts. It would be the key for a deep tool support that goes beyond the pure preparation, storage, and retrieval of development documents.

We are at an exciting stage in software engineering and the integration of formal methods. A lot of the theoretical work required for the foundations has been done. What is needed is an experimental integration and application.

In the SysLab project at the Technical University of Munich we try to follow this line and make a significant effort to gain a closer relationship between formal methods and pragmatic software engineering.

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. I thank Herbert Ehler and Ursula Hinkel for a careful reading and a number of comments.

References

[Abadi, Lamport 90]
M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990

[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

[Beeck 95]
M. v. d. Beeck: 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, 1994, 128-148

[Booch 91]
G. Booch: Object Oriented Design with Applications. Benjamin Cummings, Redwood City, CA, 1991

[Broy 91]
M. Broy: Formalisation of distributed, concurrent, reactive systems. In: E.J. Neuhold, M. Paul (eds.): Formal Description of Programming Concepts. IFIP W.G. 2.2 advanced seminar, Rio de Janeiro 1989. Berlin: Springer, 1991, 319-361

[Broy 92]
M. Broy: Compositional Refinement of Interactive Systems. DIGITAL Systems Research Center, SRC 89, 1992

[Broy 95a]
M. Broy: Equations for Describing Dynamic Nets of Communicating Systems. In: E. Astesiano, G. Reggio, A. Tarlecki (eds): Recent Trends in Data Types Specification, 10th Workshop on Specification of Abstract Data Types joint with the

5th COMPASS Workshop, S.Margherita, Italy, May/June 1994 Lecture Notes in Computer Science 906, Springer 1995

[Broy 95b]
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

[Broy, Lamport 93]
M. Broy, L. Lamport: Specification Problem. http://www.research.digital.com/SRC/personal/Leslie_Lamport/dagstuhl/all.html

[Broy, Merz, Spies 96]
M. Broy, S. Merz, K. Spies (ed.): Formal Systems Specification. The RPC-Memory Specification Case Study. Lecture Notes in Computer Science 1169, 1996

[Grapes 90]
GRAPES-Referenzmanual, DOMINO, Integrierte Verfahrenstechnik. Siemens AG, Bereich Daten-und Informationstechnik 1990

[Grosu 94]
R. Grosu: A formal foundation for concurrent object-oriented programming. Dissertation, Fakultät für Informatik, Technische Universität München, December 94

[Grosu et al. 95]
R. Grosu, K. Stølen, M. Broy: A Denotational Model for Mobile Data Flow Networks. To appear

[Guttag, Horning 93]
J.V. Guttag, J.J. Horning: A Larch Shared Language Handbook. Springer 1993

[Harel 87]
D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987, 231-274

[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

[Lynch, Stark 89]
N. Lynch, E. Stark: A proof of the Kahn principle for input/output automata. Information and Computation 82, 1989, 81-92

[MSC 96]
Message Sequence Charts (MSC), Recommendation Z.120. Technical report, ITU-T, 1996

[Parnas, Madey 91]
D. L. Parnas, J. Madey: Functional Documentation for Computer Systems

Engineering (Version 2). CRL Report 237. McMaster University, Hamilton Ontario, Canada 1991

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

[SDL 88]
Specification and Description Language (SDL), Recommendation Z.100. Technical report, CCITT, 1988

[SPECTRUM 93]
M. Broy, C. Facchi, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen: 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, May 1993

[Wirsing 90]
M. Wirsing: Algebraic Specification. Handbook of Theoretical Computer Science, Vol. B, Amsterdam: North Holland 1990, 675-788