The Incremental Development of Correct Specifications for Distributed Systems

Stephan Kleuker[1] and Hermann Tjabben[2]

[1] FB Informatik, University of Oldenburg
P.O. Box 2503, 26111 Oldenburg, Germany
E-mail: kleuker@informatik.uni-oldenburg.de
[2] Philips Research Laboratories Aachen
Weißhausstraße 2, 52066 Aachen, Germany
E-mail: tjabben@pfa.philips.de

Abstract. Provably correct software can only be achieved by basing the development process on formal methods. For most industrial applications such a development never terminates because requirements change and new functionality has to be added to the system. Therefore a formal method that supports an incremental development of complex systems is required. The project CoCoN (Provably Correct Communication Networks) that is carried out jointly between Philips Research Laboratories Aachen and the University of Oldenburg takes results from the ESPRIT Basic Research Action ProCoS to show the applicability of a more formal approach to the development of correct telecommunications software. These ProCoS-methods have been adapted to support the development of extensible specifications for distributed systems. Throughout this paper our approach is exemplified by a case study how call handling software for telecommunication switching systems should be developed.

keywords: extension of existing formal methods, combination of methods, incremental development

1 Introduction

During the last few years there has been an ever increasing demand for the fast and flexible introduction of value-added services and new features into private as well as into public telecommunications networks. Intelligent networks (IN), personal communications, computer-supported telecommunications applications (CSTA) are just a few areas from which these services are emerging. Adding more and more services to the telecommunications network leads to a situation where not only the software part of the separate network components but also the structure of the network is becoming increasingly complex. Today, it is already difficult to maintain and to extend the systems. It becomes more and more difficult to understand and to predict the behaviour of the system, e.g. in situations when interactions between services occur.

Therefore it becomes a key issue to design communications system software that provably — not only arguably — meets its requirements. Aim of the project CoCoN (Provably Correct Communication Networks) is to support a stepwise

and verified development of communications systems from the requirement phase over the specification phase to an implementation. The vision that we have in mind is an engineering approach for the development of correct communications networks.

The method presented here results from the project CoCoN, carried out jointly by Philips Research Laboratories Aachen and the Department of Computer Science at the University of Oldenburg since April 1993. CoCoN is based on the ESPRIT Basic Research Action ProCoS (Provably Correct Systems) where formal methods for the design of embedded, distributed real-time systems are developed. CoCoN thus aims to show that ProCoS methods — suitably adapted — can contribute to solve problems of industrial relevance.

CoCoN extends the ideas of ProCoS [1, 2, 5] with a method for the development of *extensible* systems. An approach for the reuse and extension of proofs (produced by model-checking or an interactive verification tool) is suggested.

In this paper we identify steps of a general methodology for the incremental development of correct specifications for distributed systems. We show how several individual results and techniques can be combined to a method. Full details are suppressed in favour of an overview of the methodology. However, the formal background is outlined in 14 figures.

To illustrate the typical design steps, a system which describes a simple version of call handling is developed throughout the main text. A more detailed elaboration of our approach can be found in [14]. At the top-level view the system in our example consists of n telephones which are connected by a basic switch (see figure 1). Each call shall be represented by a different process in the network.

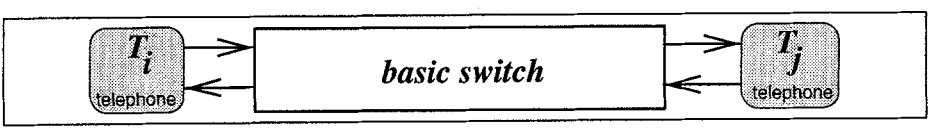


Fig. 1. Architecture of the network

The next section gives a survey of the applied methods. Section 3 describes the development of a first provably correct specification. The sections 4 and 5 describe how verified specifications can be decomposed and extended. The conclusions contain a short summary and possible further steps.

2 Approach

The main steps of the design are sketched in figure 2 and can be described as follows: The complete development begins with describing the main task of the desired system. This task is analyzed and split into subtasks. Tasks can be described in natural language. These tasks are structured as a set of informal

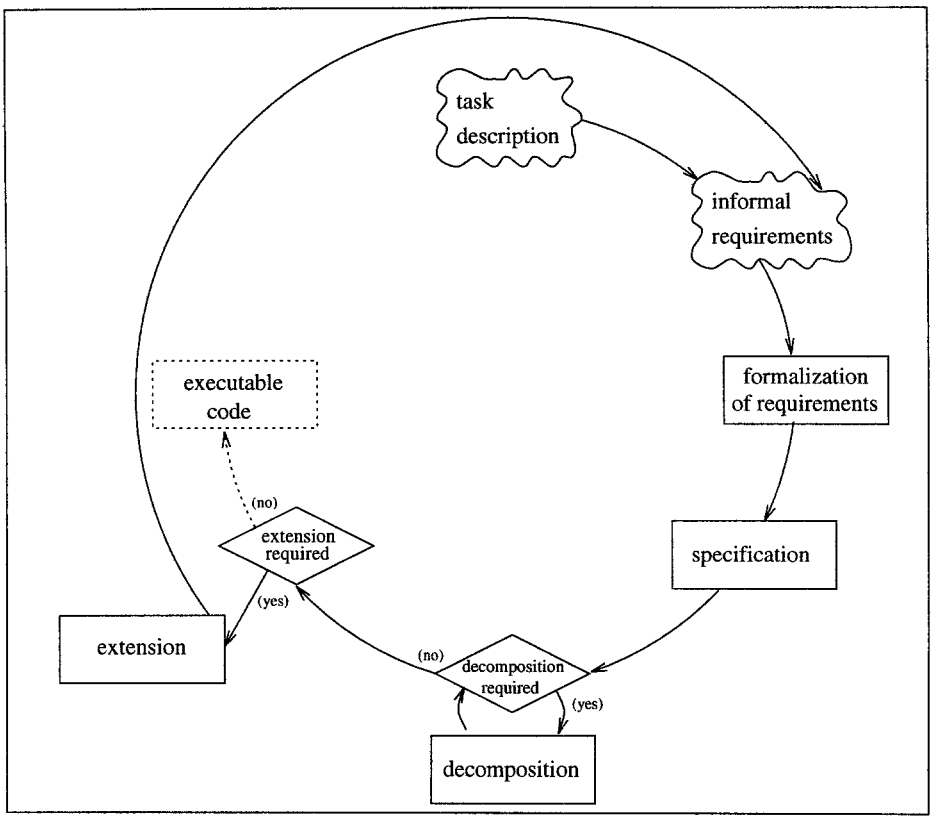


Fig. 2. Summary of the design steps

requirements. Then, informal requirements are translated into formal ones. A system (or specification) is provably correct if and only if the system fulfils these formal requirements.

The next step is the development of a first specification which already takes into account the architectural idea, i.e. it specifies the components and the interfaces between them. It is then proven that this specification fulfils the requirements and is therefore provably correct.

At this point two cases have to be considered. Either this specification is the final desired result and no changes are needed. Then this specification can be transformed into correct code. Or the specification is an intermediate result. Then, next possible steps are a decomposition of the components into sets of smaller ones or an extension of the functionality. An extension begins with an informal description of the changed behaviour of the system or one component. Typical extension tasks have the form "The following sequence of actions shall be possible, too". Here, the loop of the development from informal requirements to a verified specification begins again. This loop leads to an incremental design and therefore it is possible to start with the development of a very simple system

and to finish with a complex specification of a distributed system. Note that an extension may also include that old requirements have to be changed.

The most time consuming part of the development is the verification of the specifications against the formal requirements. Therefore a method is needed which guarantees that not each new specification has to be verified again. This is the basic idea of the so called *transformational approach.* Verified transformation rules (i.e. rules that preserve the correctness w.r.t. the requirements) are used for the system development (e.g. decomposition and extension). If such a rule is applied it is guaranteed that the result of the application fulfils the same requirements as the initial specification. Therefore, we only have to prove the application conditions of the transformation rules and need not repeat the complete verification. This approach is studied in the project ProCoS which is the foundation of CoCoN. ProCoS emphasizes a constructive approach to correctness, using verified transformations between requirements, specifications, programs and machine code.

In CoCoN two different kinds of transformation rules are used. The first ones are the fully behaviour (or semantics) preserving rules from ProCoS [29, 30]. The other transformation rules preserve only certain important requirements like deadlock-freedom. Proofs that other requirements are fulfilled have to be done again. But old proofs can be reused to a large extent because our proofs annotate the specifications. Therefore it is possible to calculate the changes that are needed for the proofs if the changes for the specifications are known.

The main reason why it is impossible to use only rules of the first kind is that we are interested in extending the system in a way that changes its behaviour. This paper demonstrates how the transformational approach is used in a stepwise design from an informal task description to the desired system.

We summarize the application of different kinds of transformation rules together with the technique of reusing proofs under the term *Specification Engineering.*

3 A First Verified Specification

3.1 From Informal Requirements to Formal ones

A correct program shall always be the final result of the development process. But, what does correct or verified mean? To be more precise: a system or specification is *correct* with respect to certain requirements, if it is *verified* that the program fulfils each requirement. So, the most important part in a formal system development is the question how to get the right formal requirements to begin with.

The following tasks are identified to come to an appropriate initial set of requirements. First, we have to specify which kind of process structure is used. This architecture is a basis for the informal requirements. Then the interfaces (set of possible communications between the processes) are fixed. The informal requirements are given in natural language and describe e.g. the interplay of

the processes. The next step is the translation of the informal requirements into formal ones. Note that the initial set of requirements is changed in the subsequent decomposition and extension steps.

It must also be taken into account that a bad set of requirements may lead to a very complex development process or, even worse, to software with undesired behaviour (if an important requirement is forgotten or formalized in the wrong way). Therefore the requirement step is the part in the development process where human faults occur most easily.

An important correctness criterion for requirements is also that of *consistency*. A set of requirements is inconsistent if it is impossible for a specification to fulfil the set. Informal requirements can be investigated by a human being whether an inconsistency exists. For formal requirements it can be proven that the requirements are consistent.

The process of finding requirements, called *requirement engineering* (see also [27]), starts with an informal description of the desired system. Any kind of description of the desired system ranging from oral descriptions to documents from related projects can be important. Requirements of the form 'if this happens then this must not (has to or might) happen' and many more have to be described. An intensive discussion is needed to come from an informal description to informal requirements. Informal requirements are simple sentences in a reduced natural language that can be understood by customers and developers. These requirements shall give a description of the initial system that we have in mind as precisely as possible. They are developed by customer and developer together.

For our example, based on the system components a first architectural concept is fixed (see e.g. figure 1) which will be decomposed into more realistic subsystems later. Our first simple system shall consist of n telephones connected by one process, called basic switch.

A typical informal requirement for our call handling system is:

- If user i dials the number of j and gets a connect signal then he or she cannot be connected with others than j.

In the next step we need to know which events are observed to formalize the requirements. Our communications are related to messages from protocols like DSS.1 (Digital Subscriber Signaling System No. 1). Table 1 lists the set of communications for the originating site (the first letter of these communications is therefore an 'O') and their informal meaning. The corresponding communications for the terminating part (starting with the letter 'T') are omitted.

The developer formalizes the requirements in a formal language which allows the verification of the derived specifications. The customer needs to understand the formal language to the extent that it can be guaranteed that customer and developer are sure of an appropriate set of requirements. A requirement language is needed that is easy to understand and in which it is possible to formalize complex parts in small formulas. We use *trace logic* [32] (traces are finite sequences of communications) as our requirement language. Trace logic is used because it is

from an originating site $T_{i_{orig}}$ to a process $Call_{i-j}$ that represents a call from i to j inside the process *network*:	
$Osetup_i$	(capital letter O for originating) initial message to the network
$Oinformation_i$	transmission of the complete number of the called party
$Odiscon_i^u$	originating site initiates call termination (^{u} for "from user")
$Odiscompl_i^n$	originating site acknowledges a call termination signal from network (indicated by ^{n})
from $Call_{i-j}$ to $T_{i_{orig}}$:	
$Oabort_i$	call is aborted by some reason like no free line or called site is busy
$Oalerting_i$	network indicates that it rings at terminating site
$Oconnect_i$	terminating site has gone off-hook
$Odiscon_i^n$	network indicates that terminating site has gone on-hook
$Odiscompl_i^u$	network acknowledges a call termination signal from originating site
From $T_{j_{term}}$ to $Call_{i-j}$ and vice versa the dual communication to the explained one.	

Table 1. Communications of the first specification

quite easy to formalize given requirements about relations between communications in such an expressive language. It is another advantage that the semantics of our specification language (introduced in a following subsection) is also based on trace logic. Verification boils down to reasoning in trace logic. An example of a formalized requirement is given in figure 3.

$$\begin{aligned}
\forall t_1, t_2, i, j \bullet\ & (([\![X]\!][t_1.(Oinformation_i, j).t_2.Oconnect_i/tr] && (1)\\
& \wedge\ t_2 \downarrow Comm(T_i) \in \{\varepsilon, Oalerting_i\}) && (2)\\
& \Rightarrow t_2 \downarrow \{Tconnect_j\} \neq \varepsilon) && (3)
\end{aligned}$$

This is a second order trace logic predicate with free variable X which stands for a simple trace predicate with one free variable tr, $[t/tr]$ denotes the substitution of tr with t, $\cdot \downarrow \cdot$ denotes the projection, $Comm(T_i)$ denotes the set of communications of T_i, ε denotes the empty word (sequence).
The variables tr, t_1 and t_2 range over traces, i.e. sequences of communications. The predicate formalizes that (1) if terminal i calls terminal j after a trace t_1 and i gets a connect signal after t_2 and (2) there is at most one alert signal in between these two communications w.r.t. terminal i then (3) terminal j has gone off-hook in between these two communications. (The ε in the second line denotes the possibility that j goes immediately off-hook without alert signal.)
The typical structure of a requirement looks like:

$$\begin{aligned}
\forall t_1, \ldots, t_k \bullet\ & (([\![X]\!][t_1.t_2.\ldots.t_k/tr]\\
& \wedge\ side_conditions(t_1, \ldots, t_k))\\
& \Rightarrow\ desired_behaviour(t_1, \ldots, t_k))
\end{aligned}$$

Fig. 3. An informal requirement formalized in trace logic

If the kind of requirements engineering as described in this paper should

be applied by engineers then it must be easy for *them* to write requirements. Trace logic uses many mathematical symbols which look at the first glance quite strange. It is necessary to change some parts of the syntax and/or to add a graphical representation to make the formulae more readable. Our approach will be a 'semi-graphical' representation to support engineers. As mentioned by Lamport [19] and others it seems to be impossible to describe each kind of set of requirements and their combination with a graphical representation. The graphics are either not powerful enough or one has to use too many different symbols. Therefore we want to choose a presentation which supports the reader to understand a requirement, but describes maybe only a subset of the expressed behaviour. For the requirement explained in figure 3 a graphical presentation can look like the diagram presented in figure 4.

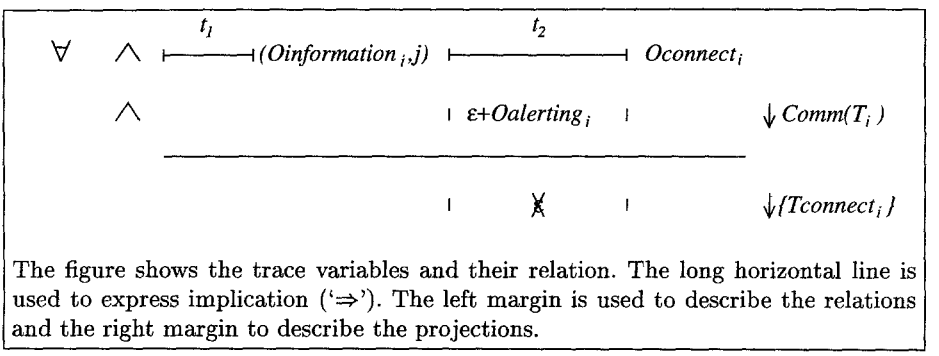


Fig. 4. Graphical representation of a requirement

3.2 First Specification

Our approach for this step can be summarized as follows: A superset of the possible system behaviour is described with *finite automata* and is reduced by examining the requirements to disable undesired behaviour. Finally, it is verified that our specification fulfils the requirements. The following text explains these steps in detail and shows how to come to a first specification for our example. It also includes an introduction to the ProCoS-specification language SL [23, 25].

We start with a description of a superset of all possible traces for all processes. Note that automata can only be used to describe a superset because their expressive power (regular languages) are not powerful enough. Finite automata (related to approaches like [11, 21]) are used to describe the behaviour of each telephone and the switch. The automata are given in figure 5. Each communication is marked to show whether it is an input ($> c$) or an output ($c >$).

Every automaton starts in its initial state, marked by an initial arrow at the top. A communication can only happen if it is possible as the next communi-

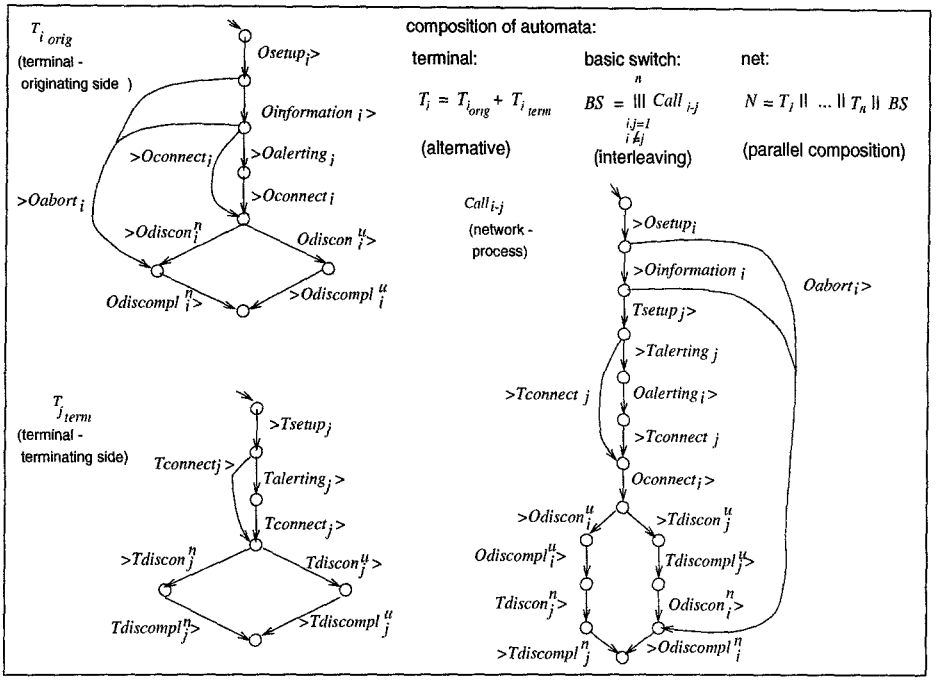


Fig. 5. Automata for each component of a call and the composition

cation by the sender and the receiver (fully synchronized communication). The automaton changes its state to the following state after performing a communication. If a process described by an automaton terminates (no communication can follow) it returns immediately to its initial state. These final states and the first state can be seen as equal or connected by an ε-arc denoting a silent transition. There is no graphical presentation of this fact in our representation because it would be the same for each automaton and because in this way we can emphasize that one cycle of the protocol has terminated.

There exists one automaton for each possible call in the specification of the network, called $Call_{i-j}$ $(i \neq j, 1 \leq i, j \leq n)$. This is a possibility to represent a dynamic number of processes (calls) in a static model.

A typical call can be described with the following sequence of actions: User i goes off-hook (an $Osetup_i$ message is sent to the basic switch). User i dials the number j (the number is transmitted with $(Oinformation_i, j)$). The terminating site is informed $(Tsetup_j)$. The switch is informed that it rings at the terminating site $(Talerting_j)$. The originating site is informed about that $(Oalerting_i)$. The terminating site goes off-hook $(Tconnect_j)$ and the originating site is informed $(Oconnect_i)$. Now, it is possible for both sites to terminate the call, e.g. the terminating site goes on-hook $(Tdiscon_j^u)$. This is acknowledged by the switch $(Tdiscompl_j^u)$. The originating site is informed $(Odiscon_i^n)$ and sends

an acknowledge message ($Odiscompl_i^n$).

The automata are composed with the following operators of process algebra (see e.g. CSP [13]):

- parallel composition ($\|$), the possibility of a trace in a parallel composition of two or more automata requires synchronization on common symbols
- interleaving ($|||$), a trace is possible in an interleaving of n automata iff it is a shuffle of traces where each is possible in one of the automata, the difference between interleaving and parallel operator is that no synchronization has to take place
- alternative ($+$), a trace is possible in an alternative iff it is possible in one of the composed automata.

Our requirements are now analyzed to determine whether there are traces possible in the parallel composition of the automata which are not allowed by the requirements. We determine from the parallel composition that it is possible that $T_{i_{orig}}$ initiates a call to $T_{j_{term}}$ but can be connected to any other telephone. The reason is that the transmitted value (the called number) is not visible in the automata and that the communication $Oinformation_i$ is one of the first communications of each automaton in the switch that describes a call with originating site T_i. The interleaving operator produces a nondeterministic choice to continue with any communication $Tsetup_x$ with $1 \leq x \leq n, i \neq x$.

The number of the terminating site j is transmitted with the communication $Oinformation_i$. The next communication w.r.t. this call shall be $Tsetup_j$. The value j has to be stored and $Tsetup_j$ activated (i.e. $Tsetup_j$ has to become the only possible next $Tsetup$ communication).

For this reason local variables are added to our specification. We then can formulate that a communication can happen only if a certain pre-condition over the local variables (an *enable-predicate*) is fulfilled. After the execution of a communication a post-condition (an *effect-predicate*) in which values of local variables may change must be fulfilled. Local variables are introduced for each process to formulate these predicates. Altogether a communication can happen if and only if

(a) it is the next possible communication of the related automata (the automata where the communication belongs to)
and
(b) the enable-predicate for this communication of each related automaton is fulfilled.

The enable- and effect-predicate for a communication are summarized in a *communication assertion*. Possible communication assertions for our example are given in figure 6.

We summarize the specification and describe it in the syntax of the specification language SL [23, 25] developed in the ProCoS project:

```
var set[1..n] of bool init false
com Oinformation_i write set
     when true then set[@Oinformation_i]'
com Tsetup_j write set[j]
    when set[j] then ¬set[j]'
```

In our example Boolean variables $set[i]$, $1 \leq i \leq n$, are used, one for each telephone inside the basic switch. Their initial values are $false$. If a communication $(Oinformation_i, j)$ happens (we refer to the communicated value as $@Oinformation_i$), the value of $set[j]$ is set to $true$. The communication $Tsetup_j$ is possible only if the value of $set[j]$ is $true$. The value of $set[j]$ is reset to $false$ after the communication $Tsetup_j$ is executed.

Fig. 6. Communication assertions

BS = spec
<interface>
<trace assertions>
<local variables>
<communication assertions>
end

The interface consists of the communications explained above together with the type of the communicated values. The trace assertions are simple regular expressions over a subset of communications of the interface. Together they describe a superset of all possible traces. An automaton is an equivalent representation of a trace assertion. Therefore we can say that the derived automata describe the trace assertions. The local variables represent the local state. They are used inside the communication assertions in the enable and effect predicates. Trace assertions, local variables and communication assertions are optional parts.

3.3 Verification

The verification that a specification fulfils the given requirements is usually the most complicated part in the formal development. The result of this effort is that verified properties needs not to be tested for the final implementation. If formal steps are applied then the correctness of the implementation w.r.t. to the formalized requirements follows immediately from the correctness of the specification. Therefore time spent with the verification is at least regained in a test phase.

Several techniques have been developed as support for a verifier. The verification of complex specifications has to be supported by computers. Two main approaches are studied in this field:

- The fully automatic approach. The specification and the requirements are the input to a computer program which checks whether the requirements are

fulfilled or not. If the requirements are not fulfilled then a counter example is presented. The advantage of this *model checking* [6, 8, 9] approach is that the verifier needs no detailed knowledge of the applied verification technique. The disadvantage is that model checking usually only works for systems with a small state space because time (and storage) which is needed for the verification usually grows exponentially in the number of components (states of the automata, numbers of variables). This *state space explosion* problems lead also to the fact that the verifier has to decide in which way he or she prepares the specification and requirements as input for the verification algorithm. This is a big restriction of the advantage mentioned before.

– The interactive approach by using an interactive verification tool like LAMDBDA [3, 4, 10]. The verifier develops a proof for a requirement step by step supported by the verification tool. The tool checks automatically whether preconditions for verification steps are fulfilled and offers suggestions for next verification steps. The verifier needs explicit knowledge about the verification techniques and needs experiences to become an effective verifier. On the other hand, it is shown in that this approach works for more complex specifications.

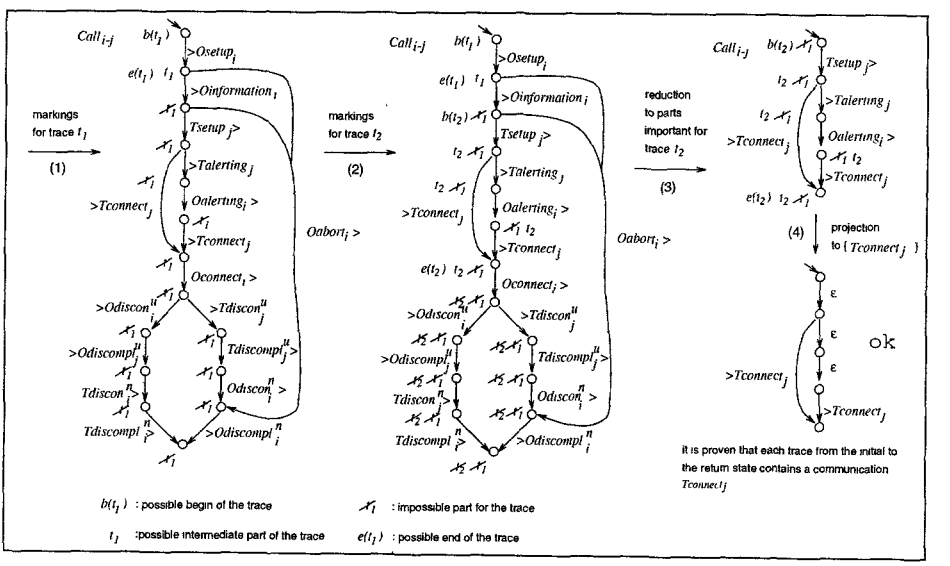


Fig. 7. Verification of the requirement in figure 3 in four steps

A third approach is the combination of the two other approaches [12, 22]. A model checker can be integrated in an interactive verification tool. The model checker solves problems of an appropriate size and the results are combined to a more complex proof.

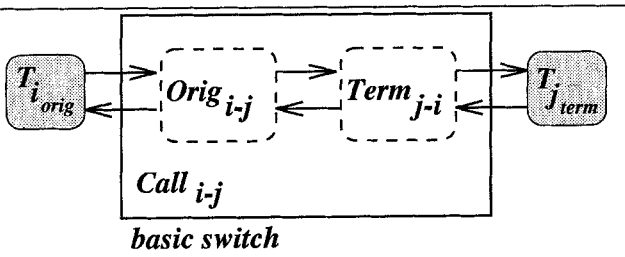
CoCoN uses the third approach. A model checking algorithm is developed which can be used for the automatic verification of a subclass of trace logic formulae. For formulae outside of this subclass the algorithm calculates the part of the specification which has to be treated an interactive verification.

A *marking algorithm* is used in which proofs annotate the states of the automata. These annotated states can be reused if the same requirement has to be checked for an extended version of the specification. The idea of marking a specification is well-known from the model-checking approach or more general from program verification [26].

The verification of the requirement in figure 3 is sketched in figure 7. Parts of a proof which are not successful do also annotate the states. This information can be used later on for the verification of extensions.

The verification follows the schema: First, it is calculated which parts of the automata can be chosen for the trace variables t_1 and t_2 of the formal requirement in figure 3. Then the side conditions and finally the desired behaviour are checked. (For full details see [18].)

4 Decomposition



A process in the switch is split into two parts: one is responsible for the originating site and the other for the terminating site in the network.

Fig. 8. Architecture after decomposition

After verifying the simple system, the question arises how to use our specification for the next steps. Our intention is to split up the network process in a distributed system where originating and terminating site are represented separately by local processes. These processes could then be allocated at different switches inside the network.

Decomposition is usually done in 3 steps:

(1) determining the *new local communications* for the interface between the desired components,
(2) *augmenting (extending)* the automaton by these local channels,
(3) *parallel decomposition* of the augmented automaton.

Suppose that in the example we add a requirement that each process $Call_{i-j}$ should consist of two parts, one related to the originating and one to the terminating site (see figure 8).

Before we decompose $Call_{i-j}$ into two separate processes we have to think about the new local communications between the new processes. We observe every state of $Call_{i-j}$ and try to find out which protocol is useful between the new processes.

$setup_{ij}$	initial message between new processes
$abort_{ij}$	for an abort of a call
$alert_{ij}$	for ringing at the terminating site
$connect_{ij}$	for a completed connection
$discon_{ij}$	for disconnect initiated
$discompl_{ij}$	for disconnect complete (acknowledge)

Table 2. Communications between originating and terminating part in the network

The interface of table 2 is introduced (a subscript ij indicates that this is a communication from site i to site j, a superscript ^{o} will indicate "from originating site" and a superscript ^{t} will indicate "from terminating site" in the termination procedure).

The new communications are still not included in the automaton $Call_{i-j}$. Therefore it is the next subtask to extend the automaton with the new communications. The decomposition of the network process is continued later.

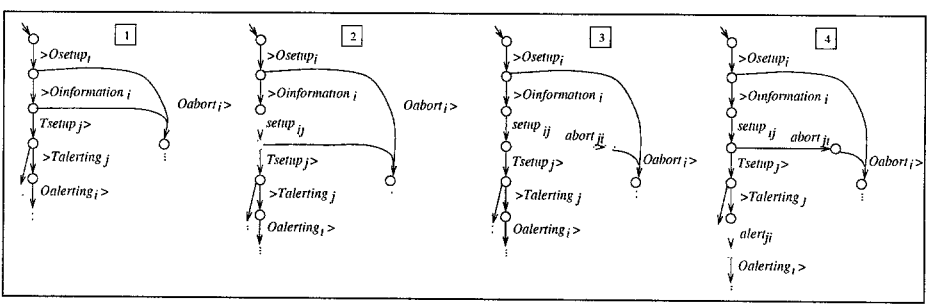


Fig. 9. Extension with local channels

The new communications are local ones. They do not have to be synchronized with the other processes. A local communication of a subprocess of a parallel composition can happen if it is a next possible communication.

The new automaton for $Call_{i-j}$ is the result of applying a transformation rule for adding new local communications several times. The first three applications are described in figure 9.

A local *setup* communication is added in part 2 of figure 9 between the transmission of the dialed number and informing the terminating site. Then, a local communication is added for a local *abort* after the *setup* communication. Finally, a local *alert* communication is added between the communication indicating that it rings at the terminating site and informing the originating site about that.

The new specification is deadlock-free because transformation rules are used which preserve deadlock-freedom. (The application criteria which have to be checked are omitted here.) The old requirements are still fulfilled because no changes are done that are relevant for these requirements.

Now, we have the possibility to use a verified semantics-preserving SL-Transformation rule to decompose the process. The processes after decomposition are described in figure 13 (with ignoring the dotted parts). The process $Call_{i-j}$ is split into $Orig_{i-j}$ and $Term_{j-i}$.

5 Extension

The previous sections described a complete path from informal requirements to a provably correct specification of a distributed system. But this specification is not likely to be a final result because the development process for large distributed systems like telephone networks never comes to an end. One important point is the extension of the existing specification. An approach is needed that takes verified specifications and the desired extensions as an input and produces a verified extension of the specification.

Let $\rightarrow_{A_i}$ be the transition relation of an automaton A_i, let A_i and A_j be two automata that are directly connected, with initial states q_{0_i} and q_{0_j}. Let q_i be a state of A_i and q_j be a state of A_j. Then q_i is in *K-relation* (K for german "Kommunikation") to q_j (abbreviated $q_i \; {}^{A_i}\mathsf{K}^{A_j} \; q_j$) iff

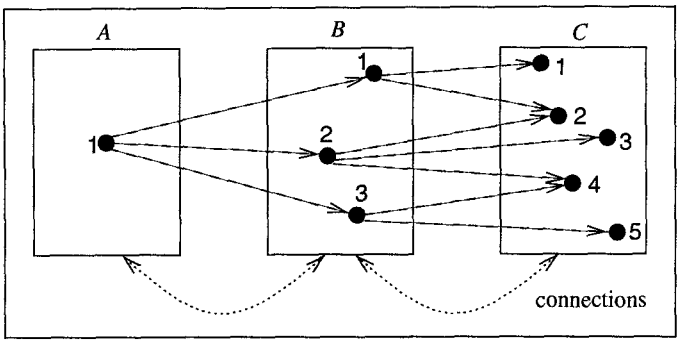
$$\exists t, t' : (q_{0_i} \xrightarrow{t}_{A_i} q_i \wedge q_{0_j} \xrightarrow{t'}_{A_j} q_j \wedge t \downarrow (Comm(A_i) \cap Comm(A_j)) = t' \downarrow (Comm(A_i) \cap Comm(A_j))$$

Informally, q_i is in K-relation to q_j iff there exists a possible trace t to q_i in A_i and a possible trace t' to q_j in A_j such that the same sequence of communications w.r.t. $Comm(A_i) \cap Comm(A_j)$ is used. Communications outside of $Comm(A_j) \cap Comm(A_i)$ can be added everywhere in t and t'.

Fig. 10. The K-relation

For the formalization of the effects of an extension and for the calculation of necessary changes an auxiliary relation between states of different processes is defined. It formalizes that if a certain subprocess is in the state p another subprocess might be in the state q (formalized in figure 10). This K-relation is used

e.g. to describe how an existing specification can be extended with preserving deadlock freedom.



The change of the behaviour of the subautomata A at state 1 can influence many states in B and C. These states are calculated by the K-relation (The dashed arrows denote K-related states, e.g. $1\ {}^{A}\mathsf{K}^{B}\ 2$). Thus a small change in one small component can have a big influence in a component that is far away from the original change.

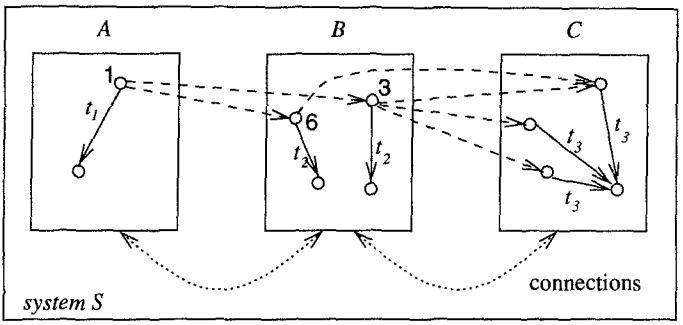
Fig. 11. A small system with an example of K-related states

Figure 11 shows an informal example for the fact that the change of the behaviour at one state can have substantial influences to many other states of the system. The system in figure 11 consists of three connected processes. The dotted arrows describe the connections between the components. Dashed arrows represent the K-relation and therefore possible states where a change in one component may lead to a different behaviour in another component. A change at state 1 of subautomaton A may lead to a new behaviour at the states $\{1,2,3,4,5\}$ of subautomaton C.

Now, we explain the extension of a system where the result is deadlock-free, too. The requirement 'deadlock freedom' is emphasized because we have observed the following: if deadlock freedom is guaranteed it can be easily shown in many cases that other requirements are fulfilled.

Let us take the example that another requirement is added to the system: we allow that the originating site can terminate a call after dialing a number. The new call termination can be described by a trace t that shall be possible in the new system. The idea is to extend each automaton A with the part of the trace which belongs to the automaton $(t \downarrow Comm(A))$.

Such a trace is added to the automaton by taking two existing states and connecting them with the new (added) trace. Then each related state of the other automata of other subprocesses is calculated. These states are also extended to make the new trace possible in the presence of synchronization and to guarantee that no new deadlocks are introduced. This idea is sketched in figure 12 and an



This figure sketches the general idea of extension. First, we choose a state (here state 1 of A) where a new trace t shall be possible, a trace $t_1 = t \downarrow Comm(A)$ is added from this state to a final state. Then, the K-related states of the extended state in B are calculated (here 3 and 6, the K-relation is painted as dashed arrows). These states are extended with $t_2 = t \downarrow Comm(B)$ to final states of B. Finally the K-related states of 3 and 6 are calculated in C and these states are extended with $t_3 = t \downarrow Comm(C)$. If S is deadlock free before the extension then it is deadlock-free afterwards, too.

Fig. 12. Extension of a distributed system with a new trace

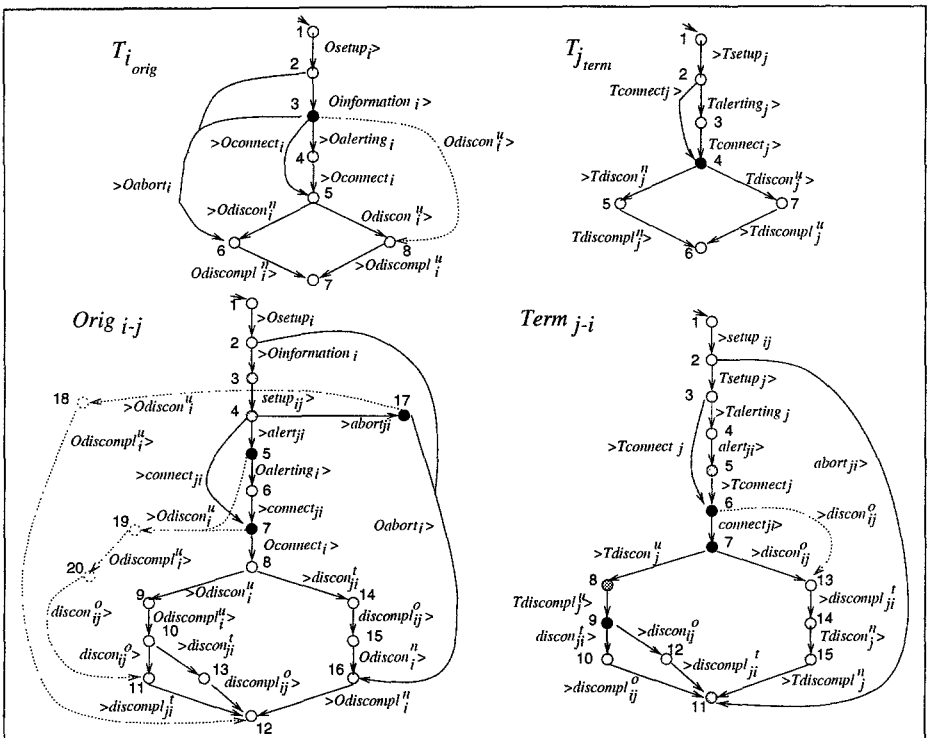
example is presented in figure 13. The task is to extend the system in such a way that it is possible for the user to go on-hook after dialing a number. The state 3 of $T_{i_{orig}}$ has to be extended. The new termination is described with a new trace which is added stepwise to the automata. The extension algorithm which is used here is described with optimizations in [16].

The new specification is deadlock free because a deadlock freedom preserving transformation rule is used. The new trace is possible because the extended state is reachable. If we want to prove the other requirements we can reuse the old proofs. The markings of the old proofs are used to calculate the markings for the added part. In most cases, the old markings need not be changed. If changes for these markings are needed, they are calculated by a back-tracking algorithm.

The idea to reuse (parts of successful) proofs is adopted from approaches for sequential programs (like [28] and related to the work of summarizing small proof steps to a large step or tactics). An example for the extension of an existing proof is given in figure 14. (The formal requirement described in figure 3 is decomposed into two requirements, the K-relation is used as auxiliary information in the proof.)

6 Conclusions and Final Remarks

The previous sections describe a general methodology based on several individual approaches for the incremental development of distributed systems. Specification

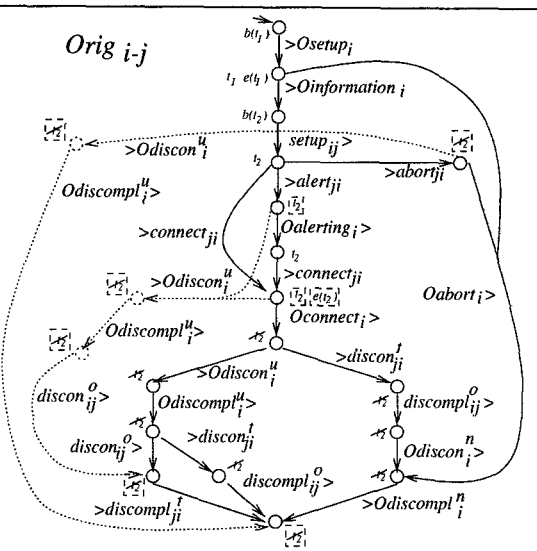


The trace $Odiscon_i^u.Odiscompl_i^u.discon_{ij}^o.discompl_{ij}^t.Tdiscon_j^n.Tdiscompl_j^n$ is added to the system. Black and gray states show K-related states that are used during the extension. The extension is described by the dotted parts. Gray states are not extended because an optimization algorithm is applied.

Fig. 13. Extension of the system

engineering is shown as a way to come to large verified specifications by small intuitive steps. It offers solutions to typical problems like system decomposition and extension of distributed systems. In contrast to other formal methods where only static systems can be developed, our approach enables us to develop extensible systems. Other approaches for an incremental design of systems like [7, 31] describe only the development of asynchronous protocols with the restriction that new communications are added one at a time.

The basic ideas of specification engineering can be transferred to other languages that are based on extended finite state machines (like LOTOS [20]). Future research will cover possibilities and limitations of this idea. Typical phases of the development of extensible systems are summarized in table 3. The way to come to a first verified specification are steps 1 and 2. An extension of a system deals with a sequence of steps 1 and 3. Note that not every typical task must be performed, e.g. in step 3 we can decide for a decomposition or an extension of



This figures shows the necessary extensions of a proof after the extension of the specifications. Markings in dashed boxed are re-calculated or new markings as a result of the marking algorithm.

Fig. 14. Extending an old proof

the functionality and if we use semantics-preserving transformation rules then no additional verification needs to be done. Otherwise requirements that are not guaranteed by the rule need to be proven again.

step	name of phase	related subjects
1	requirement engineering	informal description informal requirements formal requirements
2	initial specification	typical system behaviour superset of all possibilities restriction verification
3	specification engineering	decomposition extension of functionality transformation verification of new parts

Table 3. Phases in the development of extendable systems

We applied our approach to show the extensibility of a given Private Automatic Branch Exchange (PABX)-specification and are working on a specification

of a multi-user multimedia system [17]. By calculating which changes are needed in the existing software and whether any interaction (problems if more than one additional service is active at a time) occur the introduction of new services and features becomes much more easier.

Our case studies show that formal methods of ProCoS and CoCoN seem to be suitable for problems from the telecommunications area. Experiences of academic case studies [15, 24] can be scaled up to industrial-size problems. Nevertheless research is needed to complete each part of our method. The idea of reusing proofs has to be studied in more detail. The new transformation rules for the extension have to be rewritten for specifications with arbitrary local variables. Tools have to be built that support the proofs of requirements and the incremental development by designers.

This paper emphasizes that approaches from different areas (like the transformational approach, interactive verification tools, model-checking) must come together to build a formal method which can be used for the development of large scale extensible industrial applications.

Acknowledgments. The authors thank Martin Elixmann of Philips Research Laboratories and Ernst-Rüdiger Olderog and the other members of the ProCoS Group in Oldenburg for helpful and detailed discussions.

References

1. D. Bjørner, H. Langmaack, C.A.R. Hoare, ProCoS I Final Deliverable, ProCoS Technical Report ID/DTH db 13/1, January 1993
2. D. Bjørner et al., A ProCoS project description: ESPRIT BRA 3104, Bulletin of the EATCS, 39:60-73, 1989
3. J. Bohn, H. Hungar, TRAVERDI - Transformation and Verification of Distributed Systems, in M. Broy, S. Jähnichen, (eds.): KORSO: Methods, Languages, and Tools for the Construction of Correct Software, LNCS 1009 (Springer-Verlag), 1995
4. J. Bohn, S. Rössig, On Automatic and Interactive Design of Communicating Systems, in B. Steffen (ed.): Proc. TACAS '95, LNCS 1019 (Springer Verlag), 1995
5. J. Bowen et al., Developing Correct Systems, Bulletin of the EATCS, June 1993
6. J.R. Burch et al., Symbolic Model Checking: 10^{20} States and Beyond, in Proceedings of the Fifth Annual Logic in Computer Science, June 1990
7. D. Y. Chao, D. T. Wang, An Interactive Tool for Design, Simulation, Verification, and Synthesis of Protocols, Software - Practice and Experience, Vol. 24(8), 1994
8. E.M. Clarke et al., Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications, ACM TOPLAS 8, 1986
9. E.M. Clarke, O. Grumberg, D. Long, Verification Tools for Finite-State Concurrent Systems, in J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Decade of Concurrency, LNCS 803 (Springer-Verlag), 1995
10. M. Francis et al., LAMBDA Version 4.3, Documentation Set, 1993
11. D. Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming 8, 1987
12. H. Hungar, Combining Model Checking and Theorem Proving to Verify Parallel Processes, in C. Courcoubetis (ed.): Computer Aided Verification, LNCS 697 (Springer-Verlag), 1993

13. C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, London, 1985
14. S. Kleuker, A. Kehne, H. Tjabben, Provably Correct Communication Networks (CoCoN), Philips Research Laboratories Aachen, Technical Report, 1123/95, 1995 available by ftp: ftp.informatik.uni-oldenburg.de: /pub/procos/cocon/lab1123.ps.Z
15. S. Kleuker, Case Study: Stepwise Development of a Communication Processor using Trace Logic, in D.J.Andrews et al. (eds.): Workshop on Semantics of Specification Languages, Utrecht 1993, Workshops in Computing (Springer-Verlag), 1994
16. S. Kleuker, A Gentle Introduction to Specification Engineering Using a Case Study in Telecommunications, in P.D. Mosses, M. Nielsen, M.I. Schwartzbach (eds.): Proc. TAPSOFT '95, LNCS 915 (Springer-Verlag), 1995
17. S. Kleuker, H. Tjabben, A Formal Approach to the Development of Reliable Multi-User Multimedia Applications, in R. Gotzhein, J. Bredereke, (eds.): Proc. of the 5th GI/ITG-Fachgespräch "Formale Beschreibungstechniken für verteilte Systeme", University of Kaiserslautern, 1995
18. S. Kleuker, Model Checking with Trace Logic (Draft), University of Oldenburg, internal paper, 1995
19. L. Lamport, TLA in Pictures, technical research report, Digital Equipment Corporation, in http://www.research.digital.com/SRC/tla/,1994
20. L. Logrippo, M. Faci, M. Haj-Hussein, An Introduction to LOTOS, Computer Networks and ISDN Systems 23 (1992) 325-342, North-Holland
21. N.A. Lynch, M.R. Tuttle, An Introduction to Input/Output Automata, Technical Report CWI-Quarterly 2(3), CWI, 1989
22. O. Müller, T. Nipkow, Combining Model Checking and Deduction for I/O-Automata, in B. Steffen (ed.): Proc. TACAS '95, LNCS 1019 (Springer Verlag), 1995
23. E.-R. Olderog, Towards a Design Calculus for Communicating Programs, LNCS 527 (Springer-Verlag), p. 61-77, 1991
24. E.-R. Olderog, S. Rössig, A Case Study in Transformational Design on Concurrent Systems, in M.-C. Gaudel, J.-P. Jouannaud (eds.): Proc. TAPSOFT '93, LNCS (Springer-Verlag), 1993
25. E.-R. Olderog, S. Rössig, J. Sander, M. Schenke, ProCoS at Oldenburg: The Interface between Specification Language and OCCAM-like Programming Language. Technical Report Bericht 3/92, Univ. Oldenburg, Fachbereich Informatik, 1992.
26. S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel Programs, Acta Informatica, 16, 1976
27. H.A. Partsch, Specification and Transformation of Programs, Springer-Verlag, 1990
28. W. Reif, K. Stenzel, Reuse of Proofs in Software Verification, in Shyamasundar (ed.): Foundations of Software Technology and Theoretical Computer Science, Bombay, LNCS 761 (Springer-Verlag), 1993
29. S. Rössig, A Transformational Approach to the Design of Communicating Systems, PhD thesis, University of Oldenburg, 1994
30. S. Rössig, M. Schenke, Specification and Stepwise Development of Communicating Systems, LNCS 551 (Springer-Verlag), 1991
31. P. Zafiropulo et al., Towards Analyzing and Synthesizing Protocols, IEEE Transactions on Communications, Vol COM-28, No. 4, April 1980
32. J. Zwiers, Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes and Their Relationship, LNCS 321 (Springer-Verlag), 1989