A Gentle Introduction to Specification Engineering Using a Case Study in Telecommunications*

Stephan Kleuker**
FB Informatik
University of Oldenburg, P.O. Box 2503
26111 Oldenburg, Germany

Abstract

Software development based on formal methods is the only way to provably correct software. Therefore a method for the development of complex systems in intuitive steps is needed. A suitable solution is the transformational approach where verified semantics-preserving transformation rules are used to come from a first verified specification to the desired system. A problem is that for most industrial applications the system development never terminates because requirements change and new functionalities have to be added to the system.

This paper describes a new approach for the development of extensible specifications in small intuitive steps. New transformation rules are introduced that guarantee that intermediate results of development can be used for further steps.

keywords: extensible systems, formal software development, provably correct software

1 Introduction

Telecommunication networks are highly distributed systems, e.g. in Germany the public telecommunications network contains some thousands of switching systems, and they are required to be highly reliable. It is demanded that every switching system has an expected down time of at the most two hours in 20 years.

These switching systems have usually been built starting from some informal requirements that have changed after some time. Today, the only established way of 'proving the correctness' of a system is extensive testing. Exhaustive testing is not possible because of the complexity of software. An approach is needed for extensible, provably correct software which fulfils given requirements.

Therefore it becomes a key issue to design communication system software that provably and not only arguably meets its requirements. To come to some essential improvements of the current dissatisfying situation, the project *Provably Correct Communication Networks* — abbreviated as CoCoN — was born. CoCoN is the name of a research project carried out in close cooperation between Philips Research

*This research was supported by the Philips Research Laboratories Aachen as part of the project CoCoN (Provably Correct Communication Networks)

**E-mail: Stephan.Kleuker@informatik.uni-oldenburg.de, Tel: +49-441-798-3124

Laboratories Aachen and the Department of Computer Science at the University of Oldenburg. The overall goal of this project is to improve the software quality of future communications systems. More precisely, the aim of the project CoCoN is to support a stepwise and verified development of communication systems from the requirement phase over the specification phase to an implementation. Our method is based on results of the ESPRIT project ProCoS [3, 4, 6] (Provably Correct Systems).

ProCoS is a wide-spectrum verification project where embedded communicating systems are studied at various levels of abstraction ranging from requirements' capture over specification language and programming language down to the machine language. It emphasizes a constructive approach to correctness, using stepwise transformations between specifications, designs, programs, compilers and hardware.

But the application conditions of semantics-preserving transformation rules are often very restrictive. Therefore it is impossible to guarantee that a specification reached in a certain step of development can be transformed in any further step. For large systems lots of calculations must be done to solve this problem.

Specification engineering introduced in this paper is a new approach where each result of intermediate steps can be used without detailed information about further steps. New transformation rules are introduced in this paper that preserve only certain requirements but have less restrictive application conditions. Proofs for other requirements have to be done again where old proof structures can be reused.

Another important disadvantage of stepwise development techniques so far is that extension and change of requirements of systems are not supported.But, most systems have to be extended, i.e. they shall or must fulfil new, additional requirements not known at the time they were designed. One example is the ever increasing demand for the fast and flexible introduction of new value-added services and new features into private as well as into public telecommunications networks. Intelligent networks (IN) [1, 12], personal communications and 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 must be supported by a stepwise development of specifications.

Specification engineering can be used for the addition of new complex sequences of communications to existing systems. Other approaches for an *incremental* design of systems like [10, 22] describe only the development of asynchronous protocols with the restriction that new communications are added one at a time.

The example in the next section is an intermediate result of a stepwise development of a complete simple call handling. This result is used in CoCoN as a starting point for the introduction of correct value added services. The initial steps of a stepwise development with the transformational approach are left out here, they can be found in [14]. These stages include a development initiated by an informal description of the problem and the development of formal requirements in trace logic [23] that the system must fulfil. A specification (or program) is called correct (or verified) with respect to a set of formal requirements if it is proven that it fulfils each requirement. The complete method including the idea of specification engineering which is introduced in the following sections is sketched in the conclusions.

The next section presents a specification of a distributed call handling with finite automata. It follows an informal introduction to the extension of systems with specification engineering which leads to an extension theorem. The fourth section

discusses the application of specification engineering in a specification language which is more powerful than finite automata. The conclusions contain a short summary of the CoCoN approach and possible further steps.

2 Specification with finite automata

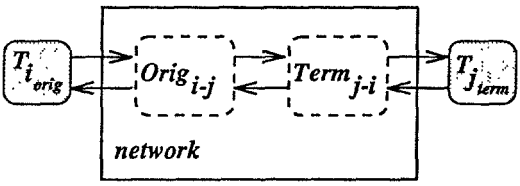


Figure 1: Example of a representation of a call from telephone T_i to telephone T_j

This section introduces a simple sort of call handling as a case study for a system that should be extensible. Finite automata are used as specification language through the following sections. Because the semantics of many specification languages is based on extended automata (or transition systems) it is possible to transfer our extension algorithm presented in the next section to other languages.

The systems in our case study are non-terminating and we assume that if a process (represented by an automaton) terminates it returns to its initial state immediately. It is possible to rewrite the following text for systems based on other types of automata.

In our example, automata are used to describe each telephone and the representation of a call in the network. A call between two sides i and j consists of four automata: $T_{i_{orig}}$ for the originating side telephone, $Orig_{i-j}$ for a representation of the originating side in the network, $Term_{j-i}$ for a representation of a call from i to j in the network and $T_{j_{term}}$ for the terminating side telephone. This situation is sketched in figure 1. Note that one telephone T_i is represented by the two automata $T_{i_{orig}}$ and $T_{i_{term}}$. This specification is an intermediate result of a development after a decomposition of the process *network*.

The interfaces (communications between the automata) and their informal meaning are given in table 1 for communications between the network and the telephone (the first letter indicates either the originating or the terminating side) and in table 2 for communications between the two processes representing a call in the network (each communication starts with a small letter, here an subscript ij is used for messages from i to j).

The specification for each process is given in figure 2 (superscript ^{o} for initiated by originating and superscript ^{t} for initiated by terminating side). Each communication is marked to show whether it is an input ($> c$) or output ($c >$). Each 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 communication by the sender and the receiver (fully synchronized communications). 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 to its initial state immediately. These final states and the first state can be seen as equal or connected by an ε-arc between them. There is no graphical presentation of this fact because it is the same for each automaton and we can emphasize that an automaton 'terminates' if no communication can follow.

from an originating side $T_{i_{orig}}$ to a process $Orig_{i-j}$ that represents a part of 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 terminating side
$Odiscon_i^u$	originating side initiates call termination (^{u} for "from user")
$Odiscompl_i^n$	originating side acknowledges a call termination signal from network (indicated by ^{n})
from $Orig_{i-j}$ to $T_{i_{orig}}$:	
$Oabort_i$	Call is aborted by some reason like no free line or called side is busy
$Oalerting_i$	network indicates that it rings at terminating side
$Oconnect_i$	terminating side has gone off-hook
$Odiscon_i^n$	network indicates that terminating side has gone on-hook
$Odiscompl_i^u$	network acknowledges a call termination signal from originating side
From $T_{j_{term}}$ to $Term_{j-i}$ and vice versa the dual communications to the explained ones.	

Table 1: Communications between network and telephone

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

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

There exist two automata for each possible call in the specification, called $Orig_{i-j}$ and $Term_{j-i}$ ($i \neq j, 1 \leq i, j \leq n$). This parametrization with i, j is a possibility to work with a dynamic number of processes (calls) in a static model.

Our first specification describes a call under the assumption that no intermediate call termination is possible. This assumption was made to arrive at a first small and easily verifiable specification.

The following definitions are used to formalize the behaviour of communicating automata (with close relation to e.g. CSP [13]).

Definition (syntax of an automaton): An *automaton* $A = (Com, Q, \delta, q_0)$ consists of four parts, a finite set Com of *communications* ($Comm(A) = Com$), a finite set Q of states ($States(A) = Q$), a partial function $\delta : Q \times Com \rightarrow Q$ which describes for a given state, and a communication the next state and the initial state $q_0 \in Q$.

A trace is possible in one automaton if there exists one sequence of states in the automaton where the connecting arcs are marked by the trace. A state where no communication can follow (termination) is called a *return state.*

Definition (possible traces and return states): A *trace* is an element of Com^*. We use t as typical letter for traces. The transition function δ is extended in the usual way from a single communication to traces (Let ε be the empty word, $\delta(q, \varepsilon) = q$, $\delta(q, t.t') = \delta(\delta(q, t), t')$, t, t' are traces). A trace t is *possible* in A iff $\delta(q_0, t)$ is defined.
The set $return(A) \subseteq States(A)$ denotes the set of states in which A immediately returns to its initial state ($return(A) = \{q \in States(A) | \forall c \in Comm(A) \bullet \delta(q, c)$ is not defined$\}$), e.g. $return(T_{i_{orig}}) = \{7\}$. The projection $\cdot \downarrow \cdot$ projects a trace on a set of communications, e.g. $a.c.b.c \downarrow \{a, b\} = a.b$.

As mentioned before there exists one automaton for each possible call in the network. Therefore we have to formalize how these automata work together to describe the whole system. The possibility of a trace in a parallel composition of two or more automata requires synchronization on common symbols and is formalized as follows:

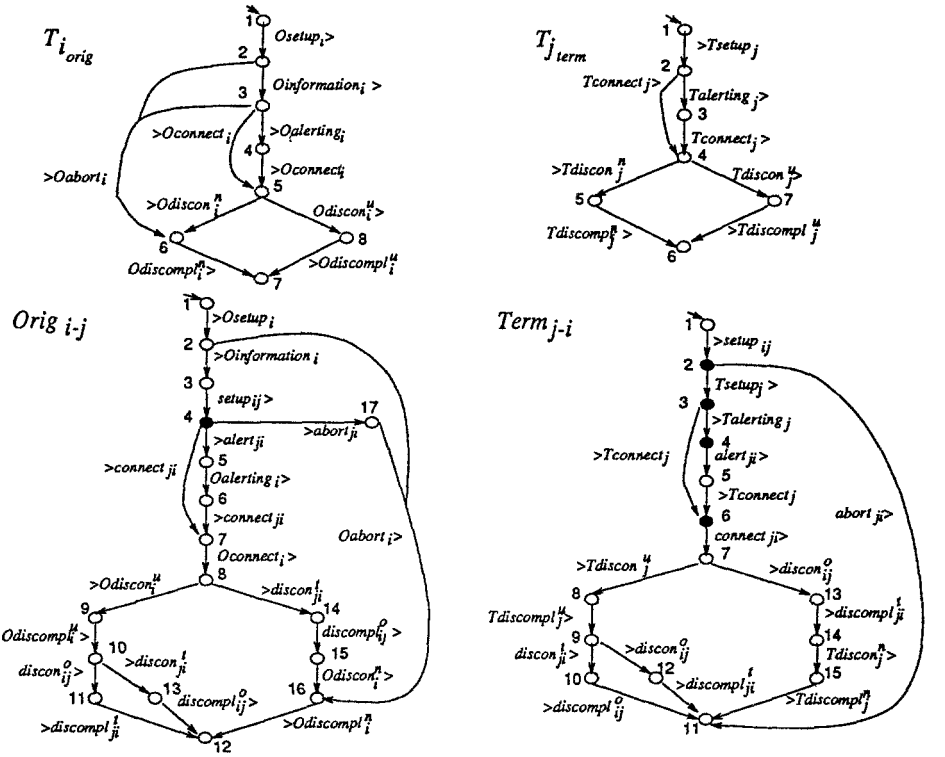


Figure 2: Specification of a distributed call

Definition (parallel composition): Let $A_i = (Com_i, Q_i, \delta_i, q_{0_i})$, $1 \leq i \leq n$ be automata. A trace $tr \in (\bigcup_{i=1}^{n} Com_i)^*$ is possible in a *parallel composition* $A_1 \| A_2 \| \ldots \| A_n$ iff $tr \downarrow Com_i$ is possible in each automaton A_i.
Formally: $\forall 1 \leq i \leq n \; \exists q_i \in Q_i \bullet \; \delta_i(q_{0_i}, tr \downarrow Com_i) = q_i$

Example: A trace $Osetup_i.Oinformation_i.setup_{ij}.alert_{ji}.Oalerting_i$ is possible in $T_{i_{orig}} \| Orig_{i-j}$, but not $Osetup_i.Oinformation_i.Oalerting_i$.

For each communication $c \in Comm(A_i) \cap Comm(A_j)$ a function $from(c)$ determines the sender and $to(c)$ of the receiver, e.g. $from(Osetup_i) = T_{i_{orig}}$, $to(Tsetup_j) = T_{j_{term}}$.

Two different processes in the network describing two different calls need not be synchronized because they are independent of each other. If we take two traces of these calls they can be mixed in any possible form to come to a new trace describing the two calls at the same time. The mixture of traces is called *merging*. The mixture of two or more automata is called *interleaving*. Therefore an interleaving operator for automata is defined. A trace is possible in an interleaving of n automata iff it is a mixture of traces each of which is possible in one of the automata. The difference between interleaving and parallel operator is that no synchronization has to take place.

Definition (interleaving): Let $A_i = (Com_i, Q_i, \delta_i, q_{0_i})$, $1 \leq i \leq n$ be automata. A trace $tr \in (\bigcup_{i=1}^{n} Com_i)^*$ is possible in an *interleaving* $A_1 \| A_2 \| \ldots \| A_n$ iff

$\forall 1 \leq i \leq n \; \exists t_1 \in Com_1, \ldots, t_n \in Com_n \bullet \; (\exists q_i \in Q_i \bullet \; \delta_i(q_{0_i}, t_i) = q_i)$
$\wedge \; tr \in merge(t_1, \ldots, t_n).$

merge is defined as: Let t_i be traces. Then

$merge(t_1, ..., t_n) = \{a_{11}.a_{21}.\ldots.a_{n1}.a_{12}.a_{22}.\ldots.a_{n2}.\ldots.a_{1k}.a_{2k}\ldots.a_{nk} |$
$a_{ij} \in Com_i \cup \{\varepsilon\} \; \wedge \; t_i = a_{i1}.\ldots.a_{ik}\}$

Example: $merge(a.b, c.d) = \{a.b.c.d, a.c.b.d, a.c.d.b, c.a.d.b, c.a.b.d, c.d.a.b\}$

A telephone is described by two automata, one for the originating and one for the terminating side. These automata are used alternatively, because in the simple call handling a telephone can either be the originating or terminating side. Therefore a third operator describes the alternative of n automata. A trace is possible in an alternative iff it is possible in one of the automata.

Definition (alternative): Let $A_i = (Com_i, Q_i, \delta_i, q_{0_i})$, $1 \leq i \leq n$ be automata. A trace $tr \in (\bigcup_{i=1}^{n} Com_i)^*$ is possible in an *alternative* $A_1 + A_2 + \ldots + A_n$ iff tr is possible in one automaton. Formally: $\exists 1 \leq i \leq n \; \exists q_i \in Q_i \bullet \; \delta_i(q_{0_i}, tr) = q_i$

Example: $Osetup_i.Oinformation_i$ is possible in $T_{i_{orig}} + T_{i_{term}}$ but not $Osetup_i.Tsetup_i$.

The underlying automata of the described processes can be summarized with the previous remarks as:

Example: Specification with finite automata.

A telephone T_i is $T_i = T_{i_{orig}} + T_{i_{term}}$. The network is $network = \prod_{i=1}^{n} \prod_{j=1, j \neq i}^{n} Call_{i-j}$.

The simple switching system is $SSS = T_1 \| \ldots \| T_n \| network$.

This specification describes all possible behaviours because each possible call and each set of possible active calls at the same time is described by the specification. Note that there are traces possible in the specification that are not desired for the final program. Therefore the automata describe a superset of the desired traces. Non-desired traces are omitted with an extension of the specification with local variables (for details see section 4).

The following definitions are used in the next section to describe the extension of systems and the resulting consequences for each subautomaton.

A relation between states of different processes is defined which is used to introduce our transformation rules. The idea is to formalize that if a certain subprocess is in the state p another subprocess might be in the state q.

Informally, q_i is in *K-relation*[1] to q_j iff it exists a possible trace t in S_i to q_i and it is possible to construct a trace t' out of t to q_j in S_j in the following way: The same communications of t w.r.t. $Comm(S_i) \cap Comm(S_j)$ have to be used to produce t' but communications of $Comm(S_j) - Comm(S_i)$ can be added anywhere in t'.

Definition (K-related states): Let $S = S_1 \| \ldots \| S_n$ be a composition of n automata, S_i, S_j $(i \neq j)$ be two parts of S that are directly connected (i.e. with $Comm(S_i) \cap Comm(S_j) \neq \emptyset$) with initial states q_{0_i} and q_{0_j}, q_i a state of S_i and q_j be a state of S_j. Then q_j is in *K-relation* to q_i (abbreviated $q_i \; {}^{S_i}\mathrm{K}^{S_j} \; q_j$) iff

$\exists t, t' \bullet \; \delta_i(q_{0_i}, t) = q_i \; \wedge \; \delta_j(q_{0_j}, t') = q_j$
$\wedge \; t \downarrow (Comm(S_i) \cap Comm(S_j)) = t' \downarrow (Comm(S_i) \cap Comm(S_j)))$

[1] K for German "Kommunikation"

Example: If we look at the processes $Orig_{i-j}$ and $Term_{j-i}$ we observe that $4\ {}^{Orig_{i-j}}\mathsf{K}^{Term_{j-i}}\ 2$, $4\ {}^{Orig_{i-j}}\mathsf{K}^{Term_{j-i}}\ 3$, $4\ {}^{Orig_{i-j}}\mathsf{K}^{Term_{j-i}}\ 4$, $4\ {}^{Orig_{i-j}}\mathsf{K}^{Term_{j-i}}\ 6$ (these states are marked black in figure 2) because after a trace $Osetup_i.Oinformation_i.setup_{ij}$ in $Orig_{i-j}$ there are some communications in $Term_{j-i}$ after $setup_{ij}$ that are independent from $Orig_{i-j}$. It follows that if $Orig_{i-j}$ is in the state 4 then $Term_{j-i}$ could be in one of the states of $\{2,3,4,6\}$.

Remark: If we observe the state space of S (the Cartesian product of the state spaces of the sub-systems) a state $(q_1,\ldots,q_n)$ can only be reached when for all $1 \leq i,j \leq n$ $q_i\ \mathsf{K}^*\ q_j$ holds, where K^* is the transitive, irreflexive closure of all K-relations.

One basic requirement which a parallel non-terminating system must fulfil is deadlock-freedom, i.e. there shall always be a possibility that a new communication can happen. The following definition of deadlock freedom is more restrictive because it ensures that after each possible trace t a new communication of each automaton of a parallel system can happen in the future.

Definition (deadlock freedom): Let $S = S_1 \parallel \ldots \parallel S_n$ be a parallel composition of automata, δ_S be the transition function and q_{0_S} be the initial state of S. Then S is called *deadlock free* iff
$\forall t \bullet (\delta_S(q_{0_S},t)$ defined $\Rightarrow (\forall 1 \leq i \leq n\ \exists t' \bullet (\delta_S(q_{0_S},t.t')$ defined $\wedge\ t' \downarrow Comm(S_i) \neq \varepsilon)))$

3 The extension algorithm

The overall idea to develop verified specifications in small steps leads to the transformational approach. Verified semantics-preserving transformation rules are used to come from a first verified specification by applying these rules to the desired system. If a specification fulfils a requirement then each result of the transformation will fulfil this requirement, too. Verified transformations are used in projects like the Munich CIP [19] and the ESPRIT basic research actions ProCoS I and II.

Case studies [6, 17] document that transformations are a suitable approach in system design. But for larger examples the question arises how to come to a first verified specification which guarantees that all desired transformation rules with their restricting application criteria can be applied in later steps. Another question is what happens to the transformational approach if system requirements are changed or new features shall be added to the system. Therefore we concentrate on a new additional technique in the following text which ensures that results of any development step can be used for further developments.

New transformation rules are added in our approach which guarantee that certain requirements are still fulfilled but can change the overall semantics. Proofs for requirements that are not guaranteed by the new rules have to be done again. But practice shows that large parts of proofs done in previous steps can be reused. The big advantage of our approach is that we come to less restrictive application conditions for the transformation rules.

We illustrate our approach with our telephone example. Suppose we wish to drop the assumption that a user cannot terminate a call at any time. We use new transformation rules of specification engineering for additional features to add possibilities of call termination.

New features are introduced by taking two states of an existing automaton of one subprocess and connecting them with a new (added) trace. Then, each related state of

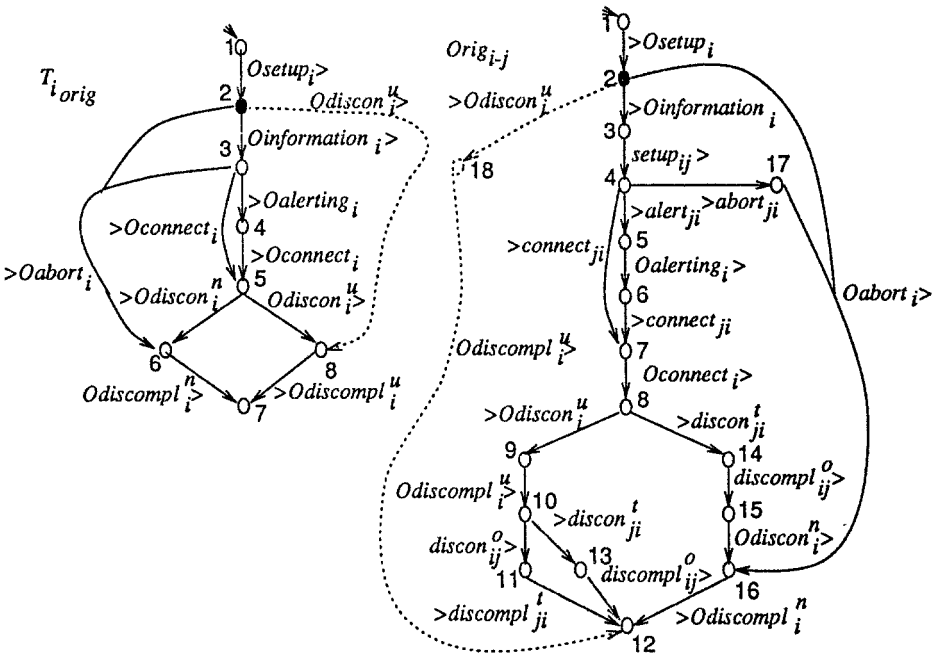


Figure 3: A first extension of the call termination at the originating side

the other automata of other subprocesses is calculated to make the new trace possible and to guarantee that no new deadlocks are introduced.

A new trace is added to the system in the following way: First, we choose a state of $T_{i_{orig}}$ where a new call termination ($Odiscon_i^u$) shall be possible. Then we calculate which states of $Orig_{i-j}$, $Term_{j-i}$ and $T_{j_{term}}$ are influenced. Finally, we have to extend the system in each calculated state with a new trace to a return state to guarantee that our system is deadlock free again. This extension is repeated for each possible call termination. We give an introduction to this idea of stepwise engineering by two extensions of our example:

Example 1: We want to add a new call termination to the originating side $T_{i_{orig}}$. A trace $t_1 = Odiscon_i^u.Odiscompl_i^u$ shall be possible in the state 2 to the return state 7 to indicate a new possible call termination. (The first communication of the new trace is the dotted arrow in $T_{i_{orig}}$ in figure 3. If it is possible to use old parts of an automaton then not the whole new trace is drawn in the automata. Instead of adding t_1 from state 2 to 7 only the trace $Odiscon_i^u$ is added from 2 to 8 because $Odiscompl_i^u$ is the only possible next communication in state 8.) We calculate for state 2 the set of K_related states of $Orig_{i-j}$ which contains only state 2. The trace t_1 is added from state 2 to the return state 12 by introducing a new state 18. Then we calculate for state 2 of $Orig_{i-j}$ the set of K_related states of $Term_{j-i}$ which contains only the initial state. Therefore nothing must be changed in $Term_{j-i}$ because this process could not 'recognize' (is not influenced) that the new trace happens. The new system is deadlock free again with an additional call termination possibility.

Example 2: We add a call-termination to state 3 of $T_{i_{orig}}$. If $T_{i_{orig}}$ reaches the

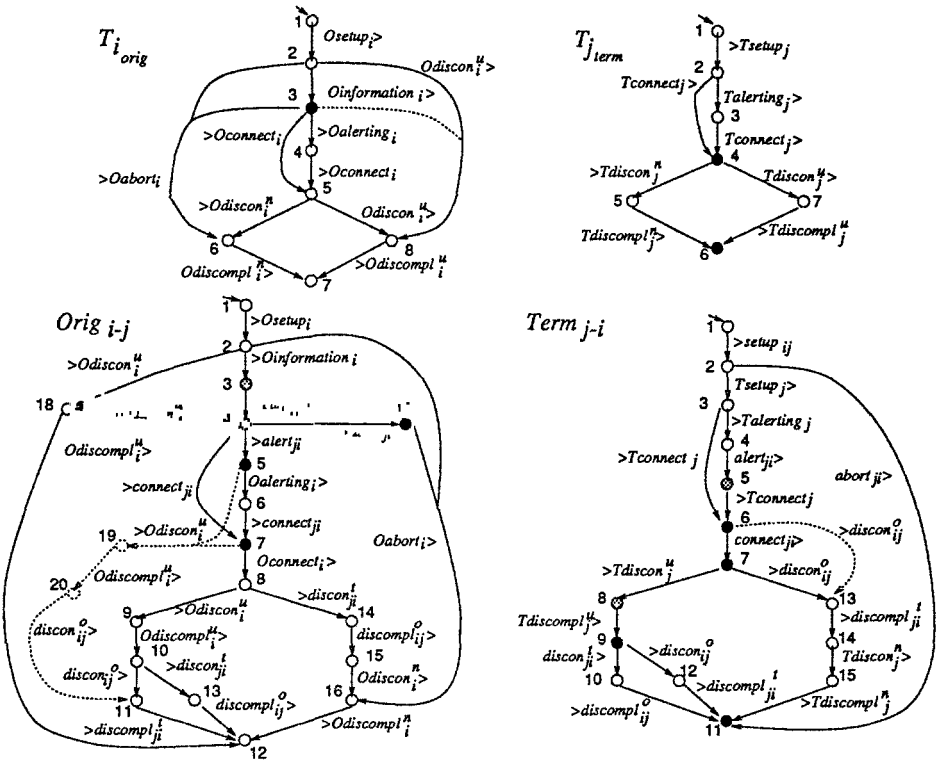


Figure 4: Another extension of the call termination

the state 3 the trace

$$t_2 = Odiscon_i^u.Odiscompl_i^u.discon_{ij}^o.discompl_{ji}^t.Tdiscon_j^n.Tdiscompl_j^n$$

describing a call termination through the system shall be possible. The extension steps are documented in figure 4. The trace $Odiscon_i^u.Odiscompl_i^u$ $(= t_2 \downarrow Comm(T_{i_{orig}}))$ connects state 3 and the return state 7 in $T_{i_{orig}}$. The set of K_related states for state 3 of $T_{i_{orig}}$ in $Orig_{i-j}$ is $\{3,4,5,7,17\}$. Following the basic idea we have to add $t_2 \downarrow Comm(Orig_{i-j})$ to each of these states. But some optimization is possible. First we can observe that in states 3 and 4 (dotted in figure 4) no communication with $T_{i_{orig}}$ can follow. Therefore communications from $T_{i_{orig}}$ can be ignored without running into deadlocks and no traces are added in the states 3 and 4.

Remark: This optimization can be added to the algorithm for calculating K_related states. Note that this optimization need not be done but leads to smaller extended systems.

For state 17 we calculate $17\ {}^{Orig_{i-j}}\mathsf{K}^{Term_{j-i}}\ 11$ as the only K_related state in $Term_{j-i}$. But 11 is a return state and therefore $Term_{i-j}$ needs no information about the new call termination in this case. Therefore only the trace $Odiscon_i^u.Odiscompl_i^u$ is added from state 17 to 12 in $Orig_{i-j}$.

The complete trace $t_2 \downarrow Comm(Orig_{i-j})$ is added to the states 5 and 7 of $Orig_{i-j}$. Next, we calculate for the states 5 and 7 the set of K_related states of $Term_{j-i}$ which is $\{5,6,7,8,9\}$. For the states 5 and 8 it holds again that no communication with the previous automaton $(Orig_{i-j})$ can follow. Therefore no trace is added for these

states. For state 9 we calculate $9\,^{Term_{j-i}}\mathsf{K}^{T_{j_{term}}}\,6$ as the only K_related state in $T_{j_{term}}$. This is a return state and therefore no information about the call termination needs to be sent to $T_{j_{term}}$. Therefore only a trace $discon^{o}_{ij}.discompl^{t}_{ji}$ is added from state 9 to 11. (This trace already exists, nothing must be changed in the automaton.) The trace $t_2 \downarrow Comm(Term_{j-i})$ is added to the states 6 and 7 of $Term_{j-i}$.

The state 4 of $T_{j_{term}}$ is the only K_related state for the states 6 and 7 of $Term_{j-i}$. The trace $t_2 \downarrow Comm(T_{j_{term}})$ is added from 4 to 6. The new system is deadlock free again with an additional call termination possibility. □

The extension ideas used in the examples are formalized in general terms. Throughout the following text $S = S_1 \parallel S_2 \parallel \ldots \parallel S_n$ will always be a parallel composition of n automata, $t = c_1.c_2.\ldots.c_m$ a trace over $\bigcup_{j=1}^{n} Comm(S_j)$, $c_1 \in Comm(S_1)$, z_1 a state of S_1. (We will define an extension of S_1 in z_1 with t. Since $\parallel$ is commutative S_1 can be an arbitrary component of S.)

Traces are added that describe one path through the system initiated in one certain state of one automaton. These traces have to fulfil certain requirements such that an extension of the automata is possible. This is formalized as follows. It exists for each trace t which is used for an extension a sequence of 'related automata'. This sequence consists of the names of the sender and the receiver of each communication. The formal definition is:

Definition (index sequence of a trace): Let be S and t be as described above. The related *index sequence* of t in S is: $s(t) := (from(c_1), to(c_1)).\ldots.(from(c_m), to(c_m))$
Example: $t = Odiscon^{u}_{i}.Odiscompl^{u}_{i}$, $s(t) = (T_{i_{orig}}, Orig_{i-j}).(Orig_{i-j}, T_{i_{orig}})$.

Sometimes it is simpler to reference the index of an automaton rather than the complete name. Now, the new example $s(t) = (1,2).(3,2).(1,3).(x,y)$ $(1 \leq, x, y, \leq n)$ is analyzed. If we extend S with a trace t (with $s(t)$ as described before) then we begin with a calculation of K-related states in S_2 of the extended state in S_1 (refering to the first pair $(1,2)$). The next calculation is done for K-related states of S_2 in S_3. The automata S_1 and S_3 are involved in the next communication (pair $(1,3)$). These automata are already extended and therefore no new calculation is needed. The next communication belongs to the automata S_x and S_y.

If $\{x,y\} \subseteq \{1,2,3\}$ then no calculation of states which have to be extended is needed. If $x \in \{1,2,3\}$ and $y \notin \{1,2,3\}$ (or $y \in \{1,2,3\}$ and $x \notin \{1,2,3\}$) then it is possible to calculate the states of S_y (S_x) which have to be extended from S_x (S_y). If $x \notin \{1,2,3\}$ and $y \notin \{1,2,3\}$ then it is impossible to calculate for the related communication which states of S_x and S_y have to be extended. There is no relation to the previous communications of t. (It must be guaranteed that $\{x,y\} \cap \{1,2,3,2,1,3\} = \{x,y\} \cap \{1,2,3\} \neq \emptyset$ holds.) Therefore such traces shall be omitted. This is done by the following definition.

Definition (traces that can be used for extensions): Let S and t be as described above. A trace t fulfils the *one-path-condition* (abbreviated $opc_S(t)$) for a system S iff

$$\forall 2 \leq j \leq m \bullet \{to(c_j), from(c_j)\} \cap (\bigcup_{k=1}^{j-1} \{to(c_k), from(c_k)\}) \neq \emptyset$$

From now on we assume $opc_S(t)$. We calculate step by step for each communication in t the set R_j of states of S_j of K_related states of the next processes that have to

be extended. A set I collects the indices of automata that have been extended. Automata in I do not have to be extended for t again because they already 'know' that the system is performing the new trace. We can describe the extension algorithm which calculates the states that have to be extended in the following way:

An algorithm for calculating the sets of states that have to be extended
Input: $S = S_1 \parallel S_2 \parallel \ldots \parallel S_n$, $t = c_1.c_2.\ldots.c_m$, $z_1 \in State(S_1)$.
Output: $R_1 \subseteq States(S_1), \ldots, R_n \subseteq States(S_n)$ sets of states of each automaton which have to be extended with a new trace (a part of t).
Internal variable: I is the set of indices of the automata which are already observed.

$R_1 := \{z_1\}$;
$I := \{1\}$;
for j= 1 to m do
  if $from(c_j) \notin I$ then [$R_{from(c_j)} :=$ K_related $(R_{to(c_j)}, S_{to(c_j)}, S_{from(c_j)})$;
    $I := I \cup \{from(c_j)\}$]
  elsif $to(c_j) \notin I$ then [$R_{to(c_j)} :=$ K_related $(R_{from(c_j)}, S_{from(c_j)}, S_{to(c_j)})$;
    $I := I \cup \{to(c_j)\}$]

Note that $from(c_j) \notin I \ \wedge \ to(c_j) \notin I$ is always false because of $opc_S(t)$. It is possible to optimize interactively (in dialogue with the specifier) the computation of K_related to determine which states should be extended.

The optimizations mentioned in the examples could happen if the K_related states are calculated. They should be done with an interactive tool which asks the user in each case whether (s)he wants an optimization. Two typical optimization criteria are: If $x\,{}^A\mathsf{K}^B\,y$ is calculated then if y is initial or return state of B or if the intersection of the next possible communications in y with $Comm(A)$ is empty then y needs not be added to the calculated set of K_related states.

It might happen that not every automaton S_j is directly related to t ($t \downarrow Comm(S_j) = \varepsilon$) but the new trace may influence S_j. Therefore we first assume that after performing the algorithm $I = \{1, 2, \ldots, n\}$ holds and present after the extension theorem a general solution.

The next definition describes an extension of an automaton, i.e. that a new trace is added between two existing states. A state z cannot be extended if the first communication of the new trace is possible in z (to preserve the determinism of δ the assumption (*) is made).

Definition (adding a trace to an automaton): Let $T = (Com, Q, \delta, q_0)$ be an automaton, $c_j \in Com$ $(1 \leq j \leq s)$, $t = c_1.\ldots.c_s$ a trace, $z \in Q - return(T)$ and $y \in return(T)$ states, (*) $\delta(z, c_1)$ is not defined,
let $\tilde{z}_1, \ldots, \tilde{z}_{s-1}$ be distinct, no elements of Q (new states used to make t possible).
The automaton $T' = (Com, Q \cup \{\tilde{z}_1, \ldots, \tilde{z}_{s-1}\}, \delta', q_0)$ is derived from S by adding the following trace to δ of T in δ': $z \xrightarrow{c_1} \tilde{z}_1 \xrightarrow{c_2} \tilde{z}_2 \ldots \tilde{z}_{s-1} \xrightarrow{c_s} y$
Then T' is called *extension* of T in z with t (short: $T' = ext(T, z, t)$).

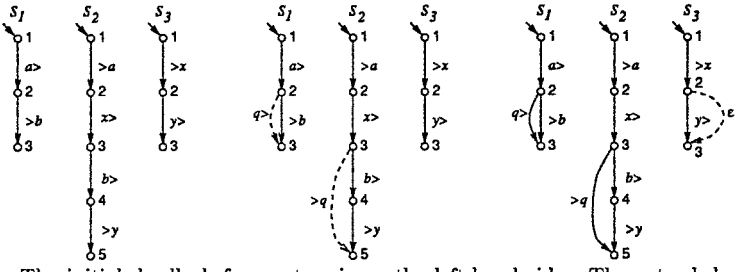
Note that y can be any return state of the automaton. It is easy to extend the definition from a state z to a set of states R by stepwise adding the trace to each state of R. The following theorem proves the correctness of a typical new transformation rule of specification engineering for the extension of distributed systems.

Extension Theorem: Let S, t, S_1, z_i be as declared above. Let R_j $(1 \leq j \leq n)$ be the sets of states computed by the extension algorithm. Let be $S'_j = ext(S_j, R_j, t \downarrow Comm(S_j))$ $(1 \leq j \leq n)$ and $S' = S'_1 \parallel S'_2 \parallel \ldots \parallel S'_n$. Then the following holds:

(i) If S is deadlock free then S' is, too.

(ii) If a trace t' is possible in S then it is possible in S', too.

Proof sketch: (i): If the new trace is initiated in S_i then each related automaton will get the possibility to work off the new trace because K_related states are extended and therefore the information about the new trace is propagated to the whole system. No mixture of old traces and new traces is possible because of the deterministic extension. The one-path-condition guarantees that there is only one initial point for the new trace and no other automaton as S_i can start the new trace. Therefore no new deadlocks are possible in S'. (ii): Traces are only added to the old automata therefore all traces of S are possible in S'.



The initial deadlock free system is on the left-hand side. The extended system (with ignoring S_3) with a deadlock after $a.x.q.a$ is in the middle. The right-hand side shows the final extended deadlock free system.

Figure 5: An extension where not every automata is directly influenced

For $I \neq \{1, 2, \ldots, n\}$ we have to calculate the related states and add a trace ε from these states to a return state. The related states are the intersection of the other K_related states. Therefore we have to compute the sets of related states of the automata S_j with $j \notin I$. These sets are (unique) solutions for the following equations:

$$\forall j \in \{1, 2, \ldots, n\} - I \bullet \quad R_j = \bigcap_{k=1}^{n} \mathsf{K\_related}(R_k, S_k, S_j)$$

A trace ε can be added to an automaton T with initial state q_o in a state z if

$$\forall c \in Comm(T) \bullet \ \delta(z, c) \text{ defined} \Rightarrow \delta(q_o, c) \text{ not defined}$$

holds. This condition guarantees a deterministic behaviour because if T is in the state z it is distinguishable whether the following communication leads from state z or from state q_0 to another state. A small additional example is given in figure 5.

The system $S_1 \parallel S_2 \parallel S_3$ is deadlock free. A trace q is added to S_1 and S_2. This trace is initiated in state 2 of S_1. From $2\,^{S_1}\mathsf{K}^{S_2}\,2$ and $2\,^{S_1}\mathsf{K}^{S_2}\,3$ follows that state 3 is extended. (State 2 needs no extension because no communication with S_1 can follow.) There is no communication w.r.t. S_3 in t but S_3 is influenced. If S_3 is not extended then the system is after the trace $a.x.q.a$ in a deadlock (S_2 is in the state 2 and S_3 in the state 2). If we do the calculations mentioned above we get: $R_1 = \{2\}, R_2 = \{3\}$, $\mathsf{K\_related}(R_1, S_1, S_3) = \{1, 2, 3\}, \mathsf{K\_related}(R_2, S_2, S_3) = \{2\}$ and $R_3 = \{1, 2, 3\} \cap \{2\} = \{2\}$. The initial communication x of S_3 is not possible in the state 2 and therefore a trace ε can be added from the state 2 to the state 1.

Many optimizations of the Extension theorem (some are mentioned in the examples) are possible. If we use a different definition of possible traces then we have to

take care of states where the new traces end (return states are a rather simple example). But this theorem is the starting point for the stepwise development of large verified and extensible specifications. Future research will lead to descriptions of the optimizations mentioned above. Note that closely related algorithms can be written to delete traces or add alternatives (or regular languages) to existing systems.

4 Applying specification engineering to a specification language

Although finite automata are useful to describe communicating processes they are not powerful enough to describe certain dependencies. Therefore we only use automata to describe a superset of all possible traces. A restriction on this set must be imposed in the next step.

The requirements developed earlier (e.g. given in [14]) are analyzed to determine if there are traces possible in the parallel composition of the automata which are not allowed by the requirements. For example, we determine from the parallel composition that it is possible that $T_{i_{orig}}$ initiates a call to $T_{j_{term}}$ but $T_{i_{orig}}$ may be connected to any other telephone. The value j is transmitted in communication $Oinformation_i$ (written as the pair $(Oinformation_i, j)$). The next communication w.r.t. this call shall be $Tsetup_j$ therefore the value j has to be stored and $Tsetup_j$ activated.

Therefore local variables are added to our specification. We can then 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*) where values of local variables may change must be fulfilled. Local variables are introduced for each process to formulate these predicates.

In our example Boolean variables $setup[i]$, $1 \leq i \leq n$, are used, one for each telephone inside the process *network*. Their initial values are $false$. If a communication $(Osetup_i, j)$ happens, the value of $setup[j]$ is set to *true*. The communication $Tsetup_j$ is possible only if the value of $setup[j]$ is *true*. The value of $setup[j]$ is $false$ after the communication $Tsetup_j$ is executed.

A *communication-assertion* is added for each communication (we refer to a communicated value which is transmitted by a communication c by writing $@c$.), e.g.:

com $Oinformation_i$ write $setup$ when $true$ then $setup[@Oinformation_i]'$
com $Tsetup_j$ write $setup$ when $setup[j]$ then $\neg setup[j]'$

Automata and communication assertions are summarized by the specification language SL [18] developed in the ProCoS. In the ProCoS project it has been shown that SL specifications can be transformed into occam-like programs [20, 21].

If we want to use our extension technique from the previous chapter, local variables have to be taken into account. Certain conditions for the enable predicates of communications of new traces must be fulfilled. Sometimes, new local variables have to be introduced to describe changes caused by a new trace. One idea is to transform the idea of superposition of UNITY [9] to SL. Further transformation rules are developed that are only possible with local variables because situations that may lead to deadlocks can be excluded by certain enable predicates over local variables. Due to lack of space only this general information can be given.

The idea of specification engineering can be used for many other specification languages whose semantics is based on transition systems. Typical examples are SDL [2] and LOTOS [15] which are widely used in the telecommunication area.

The specification engineering approach is used in SL for a stepwise development of a verified complex description of a typical call handling. This call handling is closely related to a *Basic Call State Model* [8, 11] of the ITU-T (former CCITT) standardization committee which is a suitable starting point for the development of value added services. Therefore specification engineering presents an intuitive technique for a system designer to extend specifications in small verifiable steps.

5 Conclusions and final remarks

The transformational approach of ProCoS [18, 20] with verified semantics-preserving transformation rules is extended by a new kind of transformation rules only preserving several requirements. The extension is useful because semantics-preserving rules have very restrictive application conditions and are often not suitable for system extensions.

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

Specification engineering is a way to come to large verified specifications by small intuitive steps. In contrast to other formal methods the wish for extendable systems is integrated. Basic ideas of specification engineering can be transferred to other languages based on extended finite state machines. Future research will cover possibilities and limitations of this idea.

Typical phases of the development of extensible systems in the transformational approach with specification engineering 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.

Formal proofs have to be done on computers. In ProCoS many transformation rules from SL to OCCAM are implemented and verified [5]. These rules are used in an interactive system. Tools have to be built for software engineers that support specification engineering and proofs that requirements are fulfilled. Here, reuseability of proofs will be an important part.

Next design steps will lead to a new service management process for value added services. We try to develop a simple method that explains how new services can be added to the system with a guarantee that no requirements are violated. The interplay between different services, so called *feature interaction* [7], will be one important research topic.

Acknowledgements. The author thanks M. Elixmann, A. Kehne, H. Tjabben of Philips Research Laboratories Aachen and He Jifeng, E.-R. Olderog, M. Schenke and the other members of the ProCoS Group in Oldenburg for helpful discussions.

References

[1] S. Abramowski et al., CCITT Intelligent Network Capability Set-1: Concepts and Limitations, Philips Research Laboratories Aachen, Technical Report, November 1993

[2] F. Belina, D. Hogrefe, The CCITT-Specification and Description Language SDL, Computer Networks and ISDN Systems 16 (1988/89) 311-341, North-Holland

[3] D. Bjørner, H. Langmaack, C.A.R. Hoare, ProCoS I Final Deliverable, ProCoS Technical Report ID/DTH db 13/1, January 1993

[4] D. Bjørner et al., A ProCoS project description: ESPRIT BRA 3104, Bulletin of the EATCS, 39:60-73, 1989

[5] J. Bohn, H. Hungar, Traverdi - Transformation and Verification of Distributed Systems, in M. Broy, S. Jähnichen, (eds.): KORSO, Correct Software by Formal Methods, to appear in LNCS (Springer-Verlag)

[6] J.Bowen et al., Developing Correct Systems, 5th EuroMicro Workshop on Real-Time Systems, Oulu, Finland, 1993, (IEEE Computer Society Press) 176-187

[7] E.J. Cameron et al., A Feature-Interaction Benchmark for IN and Beyond, IEEE Communications Magazine, March 1993

[8] CCITT Recommendations Q.1200: Intelligent Networks, final version, WP XI/4. Geneva. March 1992

[9] K.M.Chandy, J. Misra, Parallel Program Design, Addison-Wesley, 1988

[10] 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

[11] J.M. Duran, J. Visser, International Standards for Intelligent Networks, IEEE Communications Magazine, February 1992

[12] J.J. Garrahan et al., Intelligent Network Overview, IEEE Communications Magazine, March 1993

[13] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, London, 1985

[14] S. Kleuker, Provably Correct Communication Networks (CoCoN) (Draft Version), Philips Research Laboratories Aachen, Technical Report, 1123/95, 1995,

[15] L. Logrippo, M. Faci, M. Haj-Hussein, An Introduction to LOTOS, Computer Networks and ISDN Systems 23 (1992) 325-342, North-Holland

[16] E.-R. Olderog, Towards a Design Calculus for Communicating Programs, LNCS 527 (Springer-Verlag), p. 61-77, 1991

[17] 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

[18] E.-R. Olderog et al., ProCoS at Oldenburg: The Interface between Specification Language and OCCAM-like Programming Language. Technical Report, Bericht 3/92, Univ. Oldenburg, Fachbereich Informatik, 1992

[19] H. A. Partsch, Specification and Transformation of Programs, Springer-Verlag, 1990

[20] S. Rössig, A Transformational Approach to the Design of Communicating Systems, PhD thesis, University of Oldenburg, 1994

[21] S. Rössig, M. Schenke, Specification and Stepwise Development of Communicating Systems, LNCS 551 (Springer-Verlag), 1991

[22] P. Zafiropulo et al., Towards Analyzing and Synthesizing Protocols, IEEE Transactions on Communications, Vol COM-28, No. 4, April 1980

[23] J. Zwiers, Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes and Their Relationship, LNCS 321 (Springer-Verlag), 1989