Graphical Development of Consistent System Specifications

Bernhard Schätz *
Heinrich Hußmann ‡
Manfred Broy *

*) Technische Universität München,
Arcisstraße 21,
80333 München, Germany
Email: broy@informatik.tu-muenchen.de,
schaetz@informatik.tu-muenchen.de

‡) Siemens AG,
Public Communication Networks, Advanced Development,
Hofmannstraße 51,
81359 München, Germany
Email: hussmann@oenzl.siemens.de

Abstract. While formal methods have promised essential benefits for the software development process, industrial development reality nevertheless relies mainly on informal and especially graphical description techniques. This article argues that formal techniques are indeed useful for practical application, but they should be put to indirect use. To demonstrate this approach, two pragmatic graphical description techniques, taken from the field of telecommunication, are analyzed regarding their information content and their application in the process of specification development; as a result these techniques are formally defined. Based on the formal definition, "safe" development steps and their graphical counterparts are introduced. This yields a graphical development method which relies on precise formal foundations.

1 Introduction

Informal graphical description methods have found wide-spread application in industry. Theoreticians have often criticized these methods for their lack of a precise definition of their conveyed information. However, for industrial practice the intuitive comprehendability of graphical methods makes them well-suited for a fast development of high-quality software. Formal approaches provide a high degree of semantic preciseness. In an industrial context, nevertheless, they can be applied only to a small number of carefully selected projects with specially trained personnel. This gap between theory and practice as well as ways to overcome it has recently attracted increasing scientific attention (see e.g. [12,2,8]).

This paper presents a method that combines semantic preciseness with a pragmatic graphical notation. To ensure the practical applicability of the method, the studied graphical notation has been derived from the specifications used in an industrial development project. This work neither reports on new theoretical insights nor on a completely new graphical specification method. Instead, it is shown how the state of the art in formal methods can be applied to analyze and improve an existing specification method. This way, precision of specification can be introduced into a given software development practice while still ensuring acceptance and usability by current development personnel. So the novel aspect of the work reported here is that in this case study the protagonists of formal methods did not try to revolutionize industrial practice, but instead tried to "phase in" with existing practice and to prepare a way for smooth evolution towards more powerful specification and development methods.

This paper is structured as follows: The remainder of this introductory section describes the industrial project in the context of which this work was carried out, as well as the theoretical background of the chosen formal semantics. In section 2, the graphical description techniques are introduced which have been taken from current (non-formal) development practice in this project. Afterwards, section 3 sketches how these notations can be supported by a formal semantics. The central part of this paper is section 4, which describes the new method proposed as the result of this study. This development method uses a slightly refined variant of the original graphical notations, but it defines a number of graphical development steps which ensure consistency of the developed specifications in the sense of formal semantics. Section 5 outlines the conception of a tool basing on these development steps. Section 6 concludes the paper with an outlook to further related work.

1.1 Industrial Background

The studied graphical description methods have been taken from a functional specification used in an industrial project (at Siemens AG, Public Communication Networks, Advanced Development). The specification deals with the high-level description of a system providing an Interactive Video Service to domestic customers. The complete project (which is a system development including hardware as well as software) has a size of approximately 100 person-years (carried out in a time span of approximately 2 years).

The Interactive Video Service.

The system developed in the studied project provides domestic customers with an interactive variant of television. For this purpose, the customer's TV set is connected with a so-called set-top box, which acts as an end system for a broadband communication network based on ATM (Asynchronous Transfer Mode) and advanced switching techniques. The set-top box of the Customer Premises Equipment ("CPE") not only receives information from the network (as in classical television) but also maintains a backchannel to control program sources by commands entered by the user with an infrared remote control. Content providers ("CP") offer video material for interactive access through the communication network, using powerful server computers.

The specification used as a reference example in this paper describes a "video-on-demand" service supported by such an infrastructure, where the customer can order video information interactively for immediate delivery over the communication network (in an individual data stream for each customer).

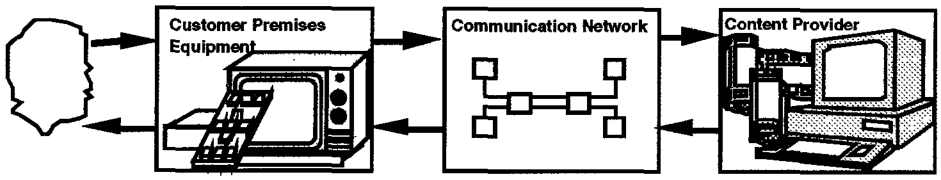

Fig. 1: The System Structure

1.2 Theoretical Background

To understand our approach requires a minimum of knowledge of the underlying formal concepts of Trace Theory. In [6], traces are introduced as a description of the system behavior:

"A communicating process is intended to interact with its environment at distinct points in time. Each individual interaction can be recorded as a value from a certain set $\mathcal{A}$ of event names (often called the alphabet of the process). An observation of the behavior of the process up to a given moment of time can be recorded as the sequence of events in which it has engaged so far. This is known as a *trace*..."

Thus, a trace can be understood as a finite sequence of symbols with each symbol denoting an action or event relevant for the system description. Given a certain set of symbols ("alphabet"), traces can be defined using the two constructors

- $\lozenge$: Denoting the empty sequence, i.e., the trace with no action occurred.
- a ⊕ t: denoting the sequence with the action "a" as its first element, and the trace "t" as its rest.

Furthermore, a function is introduced to filter out certain elements of a trace, leaving a trace consisting of parts of the original trace:

- A © t: Denoting the restriction of trace to elements from A according to the equations:

$$A \copyright \lozenge = \lozenge$$
$$A \copyright (a \oplus t) = a \oplus (A \copyright t), \text{ if } a \in A$$
$$A \copyright (a \oplus t) = A \copyright t, \text{ if } a \notin A$$

Traces will be used as a semantic basis for the representation of system behaviors.

2 Graphical Description Techniques

The following graphical description techniques are used in the specification that was the starting point of this study:

Session State Diagrams

Extended Event Traces

Please note that the specification, being an informal one, also contains a significant amount of explanatory text ("prose"). The diagrams serve merely as illustrations which are explained in the accompanying text. It was the purpose of the work described here to give a more formal, self-contained meaning to the diagrams.

2.1 Session State Diagrams

Session State Diagrams (SSDs) are used to describe the global system from the user's point of view. As implied by the name, the behavior of the system is described graphically using states. The interactions between user and system are described by transitions between those states marked with the names of these interactions. Furthermore, initial and final states are explicitly marked. Thus being similar to other classical state transition diagrams, such as Statecharts, SSDs additionally can be hierachically composed. This allows SSD to be used to describe the behavior of a system on different levels of abstraction. This is a simpler variant of hierarchical state transition systems as they are found, for instance, at the core of Statecharts [5].

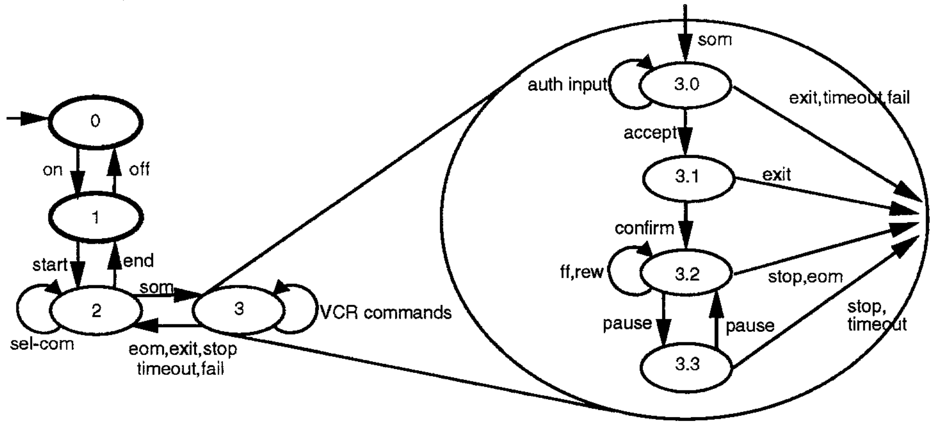

Fig. 2: System SSD and Refinement

Figure 2 describes the behavior of the "Interactive Video Service" as seen from an abstract point of view with the states "Not using IVS"(0), "Service Selection"(1), "Content Selection"(2) and "Content Transmission"(3); furthermore a more detailed view of the "Content Transmission" state is given.

2.2 Extended Event Traces

Extended Event Traces (EETs) are used for the description of the system from a more detailed, component oriented point of view. EETs describe parts from the course of interaction between two or more components; they are connected to a state of the system's SSD. In general, EETs are seen as a sample collection of legal interactions without being necessarily complete. Concerning their graphical representation and their conveyed information EETs, on the whole, correspond to "Message Flow Diagrams" or "Message Sequence Charts" (see, e.g., [9]). Additionally, EETs allow the use of indicators for repeatable or optional sequences within an EET.

Figure 3 describes the interaction of the three components "CPE", "Network" and "CP" when setting up the connection between customer and provider.

These Event Traces are called "Extended", since they go slightly beyond the well-known syntax of Sequence Charts. In particular, Extended Event Traces may contain repetition indicators (marked "- * ") to designate parts of an event trace which can appear several times in sequence, or option indicators (marked "0 -") to designate optional parts. Due to these extensions, a single EET covers a number of cases which would traditionally be depicted in several Sequence Charts. See 3.3 for a more detailed discussion of repetition indicators.

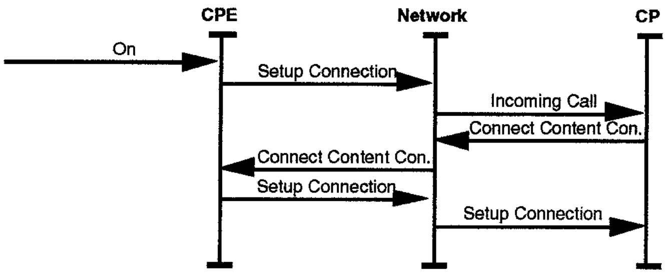

Fig. 3: EET of Connection Establishment

3 Formalization

The following section defines a formalization of the above introduced graphical notations. For the formalization of SSDs and EETs a common semantical model, the "trace model" is used (see [4] for a similar approach). This is necessary to allow the combination of both description forms into a coherent description formalism. The translation will give a precise meaning to SSDs and EETs to argue about the correctness of a development method described in section 4.

3.1 Traces by Clauses

Since SSDs and EETs will be embedded into a development method for system descriptions, it is necessary to find a coherent formal description form supporting common development mechanisms for both SSDs and EETs. As both description techniques are state based, it is reasonable to use a formalization based on states.[1] For the formal description of the system states are mapped on predicates. Each predicate characterizes the set of traces starting in the corresponding state. Transitions are represented by clauses over these predicates. Here, a simple form of clauses will be used consisting of predicates over traces. These clauses will either have the form

$P_{s1}(a \oplus t) \Leftarrow P_{s2}(t)$

denoting "An 'a' labeled transition leads from state s1 to state s2"[2], or the form

$P_{s1}(t) \Leftarrow P_{s2}(t)$

denoting "An unlabled transition leads from state s1 to states2".[3] Now, the behavior of the system given by such a set of clauses can be defined to be the set of traces characterized by those predicates that meet the conditions given by the corresponding clause set. Since several different predicates may be possible solutions for a clause set of this form, the strongest ("closest to false") solution will be chosen according to the *closed world assumption* (see, for instance, [10]).[4]

[1] While labelled transition systems (see, e.g., [11]) are an alternative representation, we chose trace semantics because of the simple refinement notion that will be used in section 4.5.

[2] Note that the orientations of the transition and of „$\Leftarrow$" are reversed.

[3] Here, „$\Leftarrow$" denotes the reverse implication.

[4] Otherwise, the trivial solution **True** would always fulfill the requirements for the predicates, characterizing all possible traces of the alphabet.

3.2 Formal Description of SSDs

To give a formal description of SSDs, the structural elements (“state”, “transition”, “initial state”, “final state”) must be expressed in formal terms. Therefore, the constituting elements of SSDs are mapped on trace clauses in the following manner:

- A *state* is mapped to a predicate. For each state “s” a corresponding predicate “P_s” is introduced.
- A *transition* is mapped to a clause. For each transition from a state “s1” to a state “s2” labeled with action “a”, a corresponding clause “$P_{s1}(a \oplus t) \Leftarrow P_{s2}(t)$” is introduced.
- An *initial state* is mapped onto a clause relating the system predicate to the state. For each initial state “s”, a corresponding clause “$S(t) \Leftarrow P_s(t)$” is introduced, with “S(t)” denoting the predicate describing the complete system behavior.
- A *final state* is mapped onto a clause for the empty trace. For each final state “s”, a corresponding clause “$Ps(\Diamond)$” is introduced.

Table 1 shows parts of the formal description of the SSDs shown in Figure 2.

$S(t) \Leftarrow P_0(t)$	$P_2(\textbf{sel-com} \oplus t) \Leftarrow P_2(t)$
	$P_2(\textbf{timeout} \oplus t) \Leftarrow P_1(t)$
$P_0(\Diamond)$	$P_2(\textbf{som} \oplus t) \Leftarrow P_3(t)$
$P_1(\Diamond)$	
	$P_3(\textbf{eom} \oplus t) \Leftarrow P_2(t)$
$P_0(\textbf{on} \oplus t) \Leftarrow P_1(t)$	$P_3(\textbf{VCR-com} \oplus t) \Leftarrow P_3(t)$
$P_1(\textbf{off} \oplus t) \Leftarrow P_0(t)$	$P_3(\textbf{exit} \oplus t) \Leftarrow P_2(t)$

Tab. 1: Clausal Representation of Figure 3

Like in logic programming, the strongest family of predicates solving these implications is taken here.

3.3 Formal Description of EETs

To base the formal description of EETs on the same principle as of SSDs, implicit states within an EET after each interaction between two components are introduced. Figure 4 shows the EET for the “Content Selection Phase” together with the explicit depiction of those implicit states.

The mapping introduced above can be applied to EETs:

- An *implicit state* is mapped onto a predicate. For each state “i” a corresponding predicate “E_i” is introduced.
- A *transition* is mapped onto a clause. For each transition from a state “i” to a state “j” labeled with action “a”, a corresponding clause “$E_i(a \oplus t) \Leftarrow E_j(t)$” is introduced.

- An *option indicator* is mapped onto a clause connecting the beginning of the optional sequence to its end. For an optional sequence with start state "i" and final state "j", a corresponding clause "$E_i(t) \Leftarrow E_j(t)$" is introduced.[5]
- A *repetition indicator* is mapped onto a clause connecting the end of the repeatable sequence with its beginning. For a repeatable sequence with start state "i" and final state "j", a corresponding clause "$E_j(t) \Leftarrow E_i(t)$" is introduced.

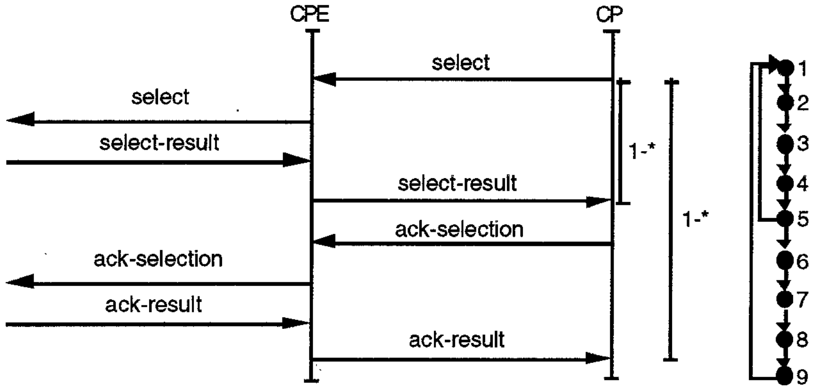

Fig. 4: EET of Content Selection and Explicit Depiction of Intermediate States

Table 2 shows the clausal description of the EET diagram depicted in Figure 4.

E_1(**select** $\oplus$ t) $\Leftarrow$ E_2(t)	E_6(**ack-selection** $\oplus$ t) $\Leftarrow$ E_7(t)
E_2(**select** $\oplus$ t) $\Leftarrow$ E_3(t)	E_7(**ack-result** $\oplus$ t) $\Leftarrow$ E_8(t)
E_3(**select-result** $\oplus$ t) $\Leftarrow$ E_4(t)	E_8(**ack-result** $\oplus$ t) $\Leftarrow$ E_9(t)
E_4(**select-result** $\oplus$ t) $\Leftarrow$ E_5(t)	E_5(t) $\Leftarrow$ E_1(t)
E_5(**ack-selection** $\oplus$ t) $\Leftarrow$ E_6(t)	E_9(t) $\Leftarrow$ E_1(t)

Tab. 2: Clausal Representation of Figure 4

4 Graphical Design Method for Consistent Specifications

It is not sufficient to simply translate the notations of an informal development method into a formal notation. The formal semantics makes sense only if some practical benefit is drawn from the introduced precision. On the other hand, the actual software developers and specifiers should be saved from direct contact with the formal notation (the clauses in our case). Therefore, we are now going to introduce a formal but graphical development technique.

Furthermore, a specification is not designed in a single step, but in several refining steps; each step is adding new details thus making an originally coarse-grained and abstract specification become sufficiently detailed. Thus, description techniques also have to be embedded into a development method, to offer a guide-line and help the developer making the necessary decision at the right time in the

[5] The formalization can be intuitively interpreted as „skipping from the start of the optional sequence to the end of it".

specification process. Such a development in general will lead to different specifications each covering different parts or aspects of the overall system. Therefore the method should support the development of "consistent" specifications, i.e., specifications which can be related in a sensible way to give rise to the complete system description without introducing contradictions or new ambiguities.

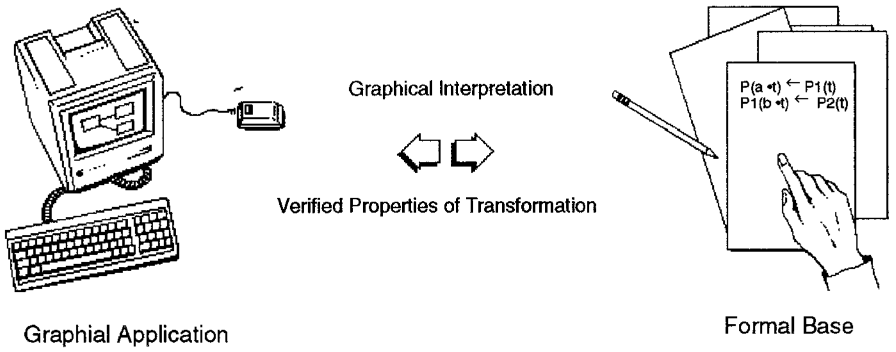

Fig. 5: Indirect Use of Formalization

For industrial use, two features of this method are most important, which are the *transparent use of formal methods*, and the *structured development of system descriptions*. In the following, after a short introduction to these concepts, the steps of this method are described. For the development of an SSD description, a more complex example will be given. Furthermore, the formal properties of the introduced transformations will be discussed. A possible conception for a corresponding development tool will close this section.

4.1 Transparent Use of Formal Methods

As argued in the introduction neither the informal nor the formal approach in its pure form is apt for industrial use, and only a combination of both aspects is adequate for the engineering process. The basic idea for combining the advantages of formal and graphical approaches is to use formality in a *transparent* way without the user's notice. One possible way for such an approach is the indirect use of formal description transformation ("refinement") rules. Here, the user of the method (the software developer) uses a fixed set of transformation rules for the derivation of graphical system descriptions (such as SSDs and EETs). Independently of this practical application, and once for all developments, a semantics specialist has used the formal interpretation of the graphical description techniques to prove that this set of transformation rules is "safe", in the sense that each transformation results in a formal refinement step[6]. This way, a "graphical development calculus" is introduced with verified "safe" operations on the description graphs.

This approach has advantages over an informal development of graphical specifications as well as over a purely formal development. The key point is that the notion of consistency of a - possibly compound - specification is well-defined, based on the

[6] The formal definition of the term „refinement" as used in our approach will be given in section 4.5.

formal semantics.[7] Moreover, a methodical guideline for the development of graphical specifications is offered, where the consistency is ensured step-by-step during the development. The next section describes this structured development method in more detail.

4.2 Development of System Descriptions

In 4.1 the design process was claimed to profit from restricting the legal design steps to controlled transformations according to the "graphical calculus". Furthermore, the use of the description techniques should be regulated to further the development process.

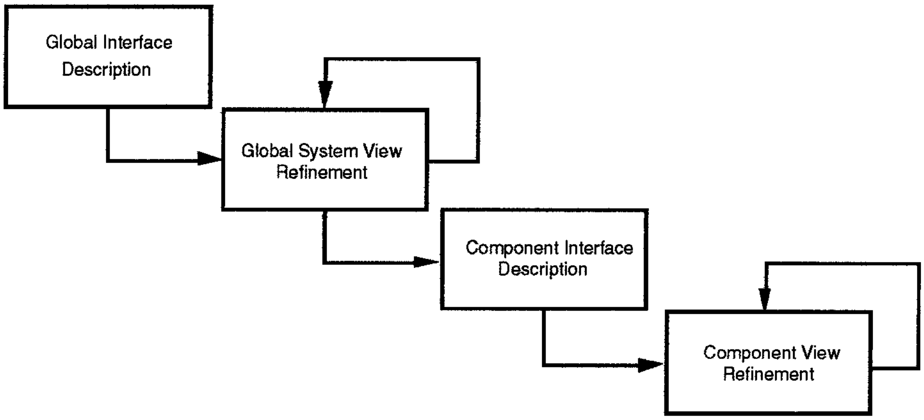
Global Interface Description
Global System View Refinement
Component Interface Description
Component View Refinement

Fig. 6: The System Description Development Process

Thus, it is necessary to embed these description techniques in a structured development guideline to facilitate the design decisions by a clear separation of concerns during the development process. To meet this requirement, the description techniques are embedded in a development guideline based on the structured process found in formal specification development, consisting of four phases as depicted in Figure 6:

- Declaration of the syntactic interface of the global system,
- Refinement of the global view system description,
- Declaration of the syntactic interface of the components,
- Refinement of the component view system description.

A similar approach is used in other design methodologies, such as FOCUS [1]. The remainder of this section will deal with these development steps. Since it is obvious that such a straight-forward development process will in general not be possible in real world projects, sufficient support must be given for the revision of system descriptions. Section 5 will elaborate this question in more detail.

[7] In our approach, the consistency between a prior specification and the newly added details is defined in terms of a refinement relation between the original specification and the complete, more detailed specification.

4.3 The Global Syntactic Interface

To describe the syntactic interface of the system, the set of relevant interactions between system and system environment/user has to be defined. Furthermore for each interaction it has to be determined whether the action is controlled by the system or the system user. Actions controlled by the system are considered to be *output actions*, while actions controlled by the user are considered to be *input actions*. In the case of the Interactive Video Service, the set of input actions contains **Start Session, EXIT, FF, PLAY, REW, EXIT**, while the set of output actions contains actions like **End of Movie** or **End of Session**.

By declaring the syntactic interface of the global system, the description of the behavior of the most "liberal" or maximally underspecified system is also determined. This system description, called the *initial description* characterizes a system allowing arbitrary interactions between the system and the environment. The initial description is the starting point from which the final description will be deduced by repeated refinement in the following development steps.

4.4 Consistent Transformation of SSDs

During the repeated refinement steps a sufficiently detailed description of the global system starting from the initial description is developed by adding more design decisions. Each refinement step consist of the application of a transformation rule. For the transformation of SSD descriptions two rules are defined:

- State splitting
- Transition elimination.

While this rule set is *elementary* and consists of simple rules, it is, on the other hand, *complete* and thus allows the deduction of an arbitrary SSD beginning with the initial description.

State Splitting.

The first kind of transformation allows the introduction of new states by splitting an already existing state in two new states. The transitions of the two new states are determined in the following way.

- For any transition starting at the old state two corresponding transitions starting at each of the new states is introduced.
- For any transition ending at the old state two corresponding transitions ending at each of the new states is introduced.
- For any transition starting and ending at the old state two corresponding transitions starting and ending at the new states are introduced.
- For any transitions starting and ending at the old state two corresponding transitions starting at each one of the new states and ending at the other are introduced.

The following scheme considers incoming ("a1", "a2") and outgoing ("e1", "e2") transitions as well as feed-back ("i") transitions of the state "s" to be split.

$P_{r1}(a1 \oplus t) \Leftarrow P_s(t)$

$P_{r2}(a2 \oplus t) \Leftarrow P_s(t)$

$P_s(i \oplus t) \Leftarrow P_s(t)$

$P_s(e1 \oplus t) \Leftarrow P_{t1}(t)$

$P_s(e2 \oplus t) \Leftarrow P_{t2}(t)$

First, for each of the generated states s1 and s2, the corresponding transitions are introduced, by replacing s by s1 and s2, respectively, throughout the clauses for s.

$P_{r1}(a1 \oplus t) \Leftarrow P_{s1}(t)$	$P_{r1}(a1 \oplus t) \Leftarrow P_{s2}(t)$
$P_{r2}(a2 \oplus t) \Leftarrow P_{s1}(t)$	$P_{r2}(a2 \oplus t) \Leftarrow P_{s2}(t)$
$P_{s1}(i \oplus t) \Leftarrow P_{s1}(t)$	$P_{s2}(i \oplus t) \Leftarrow P_{s2}(t)$
$P_{s1}(e2 \oplus t) \Leftarrow P_{t2}(t)$	$P_{s2}(e1 \oplus t) \Leftarrow P_{t1}(t)$
$P_{s1}(e1 \oplus t) \Leftarrow P_{t1}(t)$	$P_{s2}(e2 \oplus t) \Leftarrow P_{t2}(t)$

Furthermore, for each transition originally leading back into the split state, a corresponding transition from the first to the second state and vice versa is introduced.

$P_{s1}(i \oplus t) \Leftarrow P_{s2}(t)$

$P_{s2}(i \oplus t) \Leftarrow P_{s1}(t)$

Figure 7 shows the splitting of state "s" in the corresponding graphical description.[8]

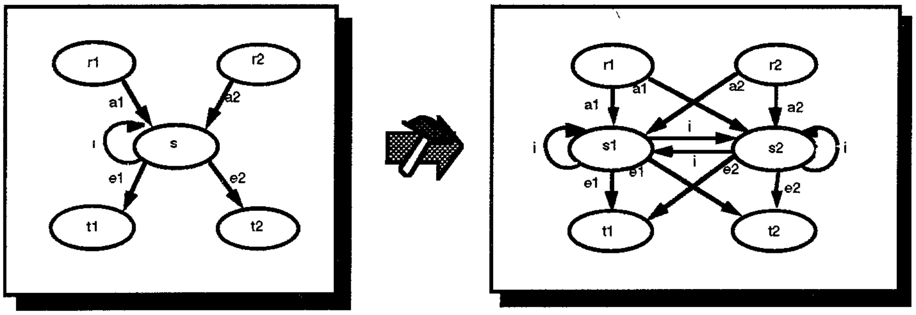

Fig. 7: Splitting of a State

Transition Elimination.

Although state splitting yields an additional structuring of the set of states, it does not restrict the set of possible system behaviors. Since the behavior of the system is essentially determined by the set of transitions, a possibility to reduce those transitions has to be introduced. This is done by simply removing a clause, in our example, for instance, the clause

$P_{s1}(e1 \oplus t) \Leftarrow P_{s2}(t)$

from the set of clauses describing the system. To restrict the behavior of a simple system of two states and three transitions, the set of clauses

$P_{s1}(e1 \oplus t) \Leftarrow P_{s2}(t)$

$P_{s1}(e2 \oplus t) \Leftarrow P_{s2}(t)$

$P_{s2}(e3 \oplus t) \Leftarrow P_{s1}(t)$

[8] The „hammer arrow" is used to indicate transformation steps which can be carried out mechanically.

can be reduced to the set

$P_{s1}(e2 \oplus t) \Leftarrow P_{s2}(t)$

$P_{s2}(e3 \oplus t) \Leftarrow P_{s1}(t)$

Figure 8 shows the same transformation on the graphical description level.

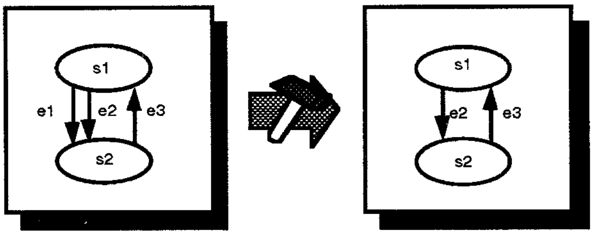

Fig. 8: Elimination of a Transition

Since not every set of states and transitions describes a "reasonable" system, the possibility of eliminating transitions should be restricted. In case of interactive systems "reasonable" means, that input actions should never be inhibited, but only be ignored. For this kind of systems the following restriction is appropriate:

"A transition labeled with an input event can only be eliminated if there is at least one other transition with the same label originating from the same state."

An Example.

The example depicted in Figure 9 demonstrates the use of the graphical transformation rules for the development of the on-line video transmission control occurring during the interactive video service. At the first level of abstraction, the influence of the control commands (**PLAY,...REW**) is left open. They show no distinct behavior since they leave the state unchanged.

The second level introduces a "pause"-state by splitting off a new state and elimination the appropriate transitions.[9] This leads to a more differentiated behavior of the system defining the system's reaction given a **PAUSE** command including the ways to leave this "pause"-state.

Finally, the original state is split again, yielding a "stop"-state together with the corresponding transition eliminations. Again, this leads to a more detailed description of the system behavior clarifying underspecified questions of the interaction.

What looks obvious in this small example may become much harder to be checked in larger development steps: the transmission control can only be entered and exited by the originally defined actions making the refined specification fit in the overall specification; furthermore for each user interaction there is a well-defined system interaction yielding a precise system description.

4.5 Formal Properties of SSD Transformation

The upcoming proofs of the acclaimed propositions are only proof sketches to illustrate the proof ideas. For these sketches, *clause schemes* will be used for illustra-

[9] The eliminated interactions are depicted in a shaded font.

tion. Those clause schemes are terms with state predicates as their free variables. They are those terms that are build by conjunction of those clauses used to describe SSDs and EETs according to section 3. Thus, e.g., the conjunction of the clauses in Table 1 yields a clause scheme with free variables S (describing the system behavior), and $P_0,..,P_4$.

Since the clauses used here are all positive clauses, for a sufficient interpretation the *closed world assumption* has to be used, associating the *strongest* possible predicate P as solution of a term of the form

$$\exists\, P_{s1} \dots P_{sn}.\ C(P, P_{s1},\dots, P_{sn})$$

Here, P stands for the predicate characterizing the system behavior, $P_{s1},\dots,P_{sn}$ are the state predicates denoting those states used in the formal description of the system, and

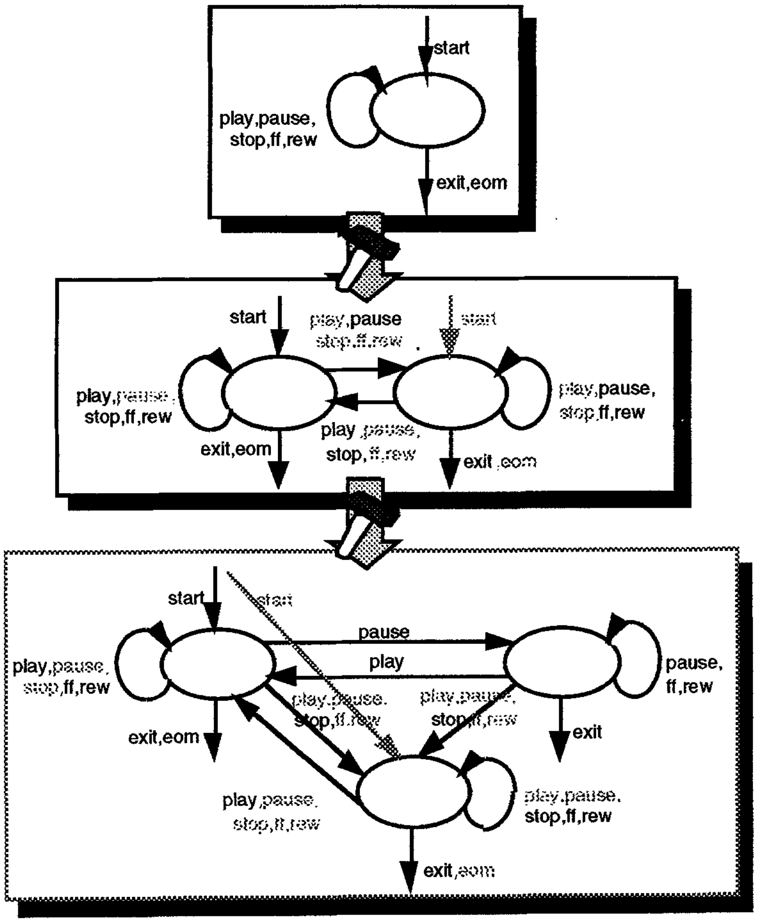

Fig. 9: Example Refinement of SSDs

C stands for the conjunction of the set of clauses used to described the system behavior.

State Splitting.

As already mentioned in 4.4, transformation by splitting a state si into states si1 and si2 does *not* change the behavior of the described system. More formally, it leaves the set of characterized traces unchanged.[10] According to the structure of the transformation, the clause set including the head clause is reorganized into the conjunction of the clause schemes N and F; here, F denotes the feedback clauses of the state to be split, i.e., those clauses describing transition with si as starting and ending state. N denotes the set of the remaining clauses. Using these abbreviations, state splitting can be seen as substituting the clause scheme

$N(P,P_{s1},...,P_{si},...,P_{sn}) \wedge F(P_{si},P_{si})$

with the scheme

$N(P,P_{s1},...,P_{si1},...,P_{sn}) \wedge F(P_{si1},P_{si1}) \wedge$

$N(P,P_{s1},...,P_{si2},...,P_{sn}) \wedge F(P_{si2},P_{si2}) \wedge$

$F(P_{si1},P_{si2}) \wedge F(P_{si2},P_{si1}).$

Thus, the main part of the proof consists of showing that

$$(\exists\, P_{s1} \ldots P_{si1}\ P_{si2} \ldots P_{sn}.\ C(P,P_{s1},...,P_{si1},P_{si2},...,P_{sn})) \Leftrightarrow$$
$$(\exists\, P_{s1} \ldots P_{si} \ldots P_{sn}.\ C(P,P_{s1},...,P_{si},...,P_{sn}))$$

"$\Rightarrow$" follows immediately by choosing either P_{si1} or P_{si2} for P_{si}. "$\Leftarrow$" follows by choosing P_{si} for both P_{si1} and P_{si2}.

Transition Elimination.

While state splitting leaves the system behavior and the corresponding trace set unchanged, transition elimination does change them. Nevertheless, only a controlled change is allowed: the set of traces characterized by the refined specification is a subset of the trace of the original specification.[11] According to the transformation structure of the elimination, the clause set is reorganized into the conjunction of the clause schemes N and T; T denotes the transition to be eliminated, and N the set of the remaining clauses. This abbreviation allows the transition elimination rule to be seen as substituting the clause scheme

$N(P,P_{s1},...,P_{sn}) \wedge T(P_{si},P_{sj})$

with the scheme

$N(P,P_{s1},...,P_{sn}).$

Thus, the main part of the corresponding proof consists of showing

$$(\exists\, P_{s1} \ldots P_{si} \ldots P_{sn}.\ N(P,P_{s1},...,P_{sn}) \wedge T(P_{si},P_{sj})) \Rightarrow$$
$$(\exists\, P_{s1} \ldots P_{si} \ldots P_{sn}.\ N(P,P_{s1},...,P_{sn})),$$

which trivially holds.

[10] This equivalence relation, a stronger notion of refinement than usually used, between two specifications (i.e.,predicates) S_1 and S_2 can be mathematically expressed as „$S_1 \Leftrightarrow S_2$".

[11] This relation, corresponding to the usual refinement notion used with traces, between a specification (i.e., predicate) S_1 and S_2 can be mathematically expressed as „$S_2 \Rightarrow S_1$".

4.6 Syntactic Interfaces of Components

Like the development of the description of the global system, the development of the component view starts with the determination of the syntactic interface of the components of the system. Hence, the set of relevant interactions between the components has to be determined as well as whether the actions is an input or an output action of the corresponding component.

As before, the determination of component interface again defines an initial description. To be consistent with the global description, however, the component description may not characterize arbitrary behaviors but must respect the global restrictions. Thus, arbitrary interactions between the components are only allowed within the global system states. According to the above schema, for each state s of the global system description and for each internal interaction i a corresponding clause

$$P_s(i \oplus t) \Leftarrow P_s(t)$$

is introduced. On the graphical level, this corresponds to an introduction of a feedback transition for every SSD state, labeled with all internal interactions.

Formal Property of Alphabet Change.

To change from the global system view to the component view, the set of relevant observed actions has to be changed. This transformation is referred to as "alphabet change". According to the transformation structure of the alphabet change, the clause set is reorganized into the conjunction of the clause schemes N and $F(P_{s1}) \wedge \ldots \wedge F(P_{sn})$; here, F denotes the conjunction of all newly introduced transitions labeled with internal actions i1,...,ik, and is of the form

$$P(i1 \oplus t) \Leftarrow P(t) \wedge \ldots \wedge P(ik \oplus t) \Leftarrow P(t).$$

N denotes the unchanged part of the clause set. This abbreviation allows the alphabet change to be seen as substituting the clause scheme

$$N(P_{s1}(A©.),\ldots, P_{sn}(A©.))^{12}$$

with the scheme

$$N(P_{s1},\ldots,P_{sn}) \wedge F(P_{s1}) \wedge \ldots \wedge F(P_{sn}).$$

Thus, the main part of the corresponding proof consists of showing the proposition

$$\exists P_{s1} \ldots P_{sn}.\ N(P(A©.),P_{s1}(A©.),\ldots,P_{sn}(A©.)) \Leftrightarrow$$
$$\exists P_{s1} \ldots P_{sn}.\ N(P,P_{s1},\ldots,P_{sn}) \wedge F(P_{s1}) \wedge \ldots \wedge F(P_{sn}).$$

"$\Leftarrow$" follows immediately by restricting $P_{s1},\ldots,P_{sn}$ to the alphabet A. To proof "$\Rightarrow$", i.e., to obtain the same clause scheme, $F(P_{s1}) \wedge \ldots \wedge F(P_{sn})$ has to be inferred. This follows trivially using the following deduction:

True

$\Leftrightarrow (P(A©t) \Leftarrow P(A©t))$

$\Leftrightarrow (P(A©(i1 \oplus t)) \Leftarrow P(A©t))$

[12] Here, the notation P(A©.) is used to denote the predicate Q with $\forall$ t. $Q(t) \Leftrightarrow P(A©t)$.

4.7 Refinement to Component View

The last step of the development process consists of the repeated refinement of the component view of the system. By construction, the formalizations of SSDs and EETs do not essentially differ. Therefore, from a formal point of view, no other transformations are needed for the refinement of EETs than in case of the refinement of SSDs. Since, however, the graphical description techniques differ essentially, an appropriate graphical representation of these rules must be offered. As the EETs are more restricted than SSDs concerning their expressiveness, a suitable extension of EETs should be offered for a homogeneous development method. Therefore, in this section the concept of *hierarchical EETs* is introduced, and appropriate transformation techniques are developed. Those transformation techniques are tailor-made for the communication sequence view offered by EETs in contrast to the state based view offered by SSDs and the formal description technique. Because of their similarity to the above described SSD transformation rules, the introduction of these transformations concentrates on the graphical level.

Hierarchical EETs.

As already mentioned above, EETs are much more restricted in their expressiveness compared to SSDs, since EETs do not allow to choose from a set of possible behaviors during a run of the system. To overcome this restriction EETs are enhanced by introducing a hierarchical notation and the possibility to describe the system behavior by sets of EETs.

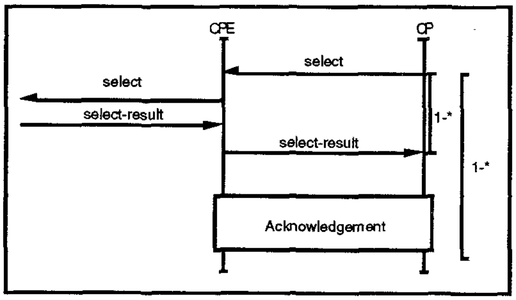

Fig. 10: Hierarchical EETs

Since an EET is already seen as a part of a larger description (i.e., the SSD), this notion is extended to make EETs legal subcomponents of another EET, too. This structuring allows hierarchical descriptions. By giving more than one EET for a certain phase of the system, each of the given possibilities becomes a legal behavior of the system. This is a standard interpretation commonly used with "Message Sequence Charts" and comparable description techniques. By combining both techniques, as suggested in current revisions of "Message Sequence Charts" (e.g., [9]) SSDs and EETs become equally expressive[13] and thus make a homogeneous description development possible, using the following three transformation rules.

[13] See [3], e.g., for the discussion of expressive power of comparable state-based formalisms.

Repetition splitting.

Those parts on an EET covered by a repetition/optionality indicator can be split into two identical copies chained to each other as depicted in Figure 11. Formally, the corresponding pairs of clauses of the form

$P_{s1}(a \oplus t) \Leftarrow P_i(t)$ and $P_f(e \oplus t) \Leftarrow P_{s2}(t)$.

are substituted by the clauses

$P_{s1}(a \oplus t) \Leftarrow P_{i1}(t)$, $P_{f1}(e \oplus t) \Leftarrow P_{i2}(t)$, and $P_{f2}(e \oplus t) \Leftarrow P_{s2}(t)$.

The corresponding repetition and optionality indicators

$P_i(t) \Leftarrow P_f(t)$ and $P_f(t) \Leftarrow P_i(t)$

are substituted by the clauses

$P_{i1}(t) \Leftarrow P_{f1}(t)$ and $P_{f1}(t) \Leftarrow P_i(1t)$,

$P_{i2}(t) \Leftarrow P_{f2}(t)$ and $P_{f2}(t) \Leftarrow P_{i2}(t)$,

as well as $P_{i1}(t) \Leftarrow P_{f2}(t)$ and $P_{f2}(t) \Leftarrow P_{i1}(t)$.

Furthermore, the corresponding part of the EET the indicator ranges over has to be split, too. This is done by substituting all the corresponding clauses

$P_r(a \oplus t) \Leftarrow P_s(t)$

by the corresponding pair of clauses

$P_{r1}(a \oplus t) \Leftarrow P_{s1}(t)$ and $P_{r2}(a \oplus t) \Leftarrow P_{s2}(t)$.

Comparable to the state splitting of SSDs, the indicator splitting does not change the behavior of the described system, but adds additional structuring.

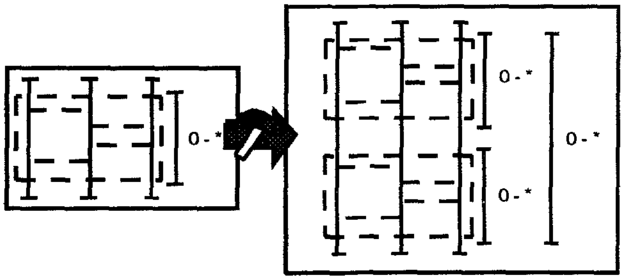

Fig. 11: Splitting of EET repetition

Subpart Splitting.

A second way of adding additional structure to the description without changing the system behavior is the splitting of subcomponents. Here, a subpart of a hierarchical system description is split up in two identical parts adding an alternative but identical behaviour.

On the level of the clausal description, the subpart splitting is carried out in similar fashion described in the case of the indicator splitting.

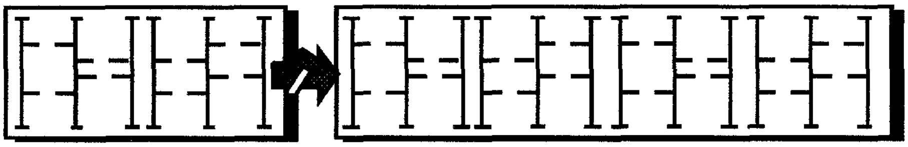
Fig. 12: Subpart Splitting

Indicator Specialization.

Finally, the indicators for repetition and optionality can be specialized in the following way:

- "0-*"-marked parts can be eliminated.
- "0-*"-marked parts can be substituted by corresponding "0-1"-marked parts.
- "0-*"-marked parts can be substituted by corresponding "1-*"-marked parts.
- "1-*"-marked parts can be substituted by corresponding unmarked parts.

All of these transformations are real behavior refinements and thus restrict the behavior of the described system.

Formal Properties.

Since all the above transformation rules given for EETs can be expressed with corresponding clause schemes as given for SSDs, the properties of the former can be deduced in equivalent fashion as the properties of the latter. For reason of brevity, the formal treatment of those rules will therefore be skipped here.

5 Conception of a Tool

As mentioned above, the introduced graphical transformation concepts for SSDs and EETs are correct, complete and elementary, and are thus good candidates for the described development process from a theoretical point of view. Nevertheless, those are not sufficient criteria in themselves for practical usefulness, which is depending essentially on proper integration in a corresponding tool. Therefore, this section will outline the conception of an appropriate tool to put this method to practical work by identifying four additional criteria:[14]

1. Although elementary transformation steps allow easy proofs of their properties, a development using these steps becomes cumbersome and is therefore of low practical use. A useful tool should hence allow us the combination of elementary steps to form more complex steps by offering "combinators" for the *creation of "macro steps"*. As indicated by example 4.4, state splitting is generally combined with transition elimination which should consequently be combined into one operation. This becomes even more obvious in the case of the development of EETs where the construction of a sequence of interactions consists of repeatedly splitting an option/repetition indicator, elimination of alternatives and the elimination of an option/repetition indicator.

[14] The next step of the cooperation is planned to include the realization of such a tool.

2. As mentioned in section 2.1, SSDs support a modular description of system behavior by allowing a refinement of SSD states by SSDs. Since descriptions of larger systems will often be developed in parallel in several groups, the envisaged tool should support such a *modularization of the system description* by offering techniques for splitting such descriptions into descriptions of subparts and re-combining them into a complete description.

3. The modular nature of SSDs also yields the possibility for views of the system behavior with different levels of abstraction. Thus this modularity is not only a useful technique for the development process, but also for the documentation and the communication of the system behavior. Thus, a suitable tool should offer a facility for the *generation of abstract views* of the system behavior by allowing to ignore irrelevant details. This may include the abstraction from certain actions, the unification of similar states, or the hiding of trivial actions and states.

4. So far, the described method only supports a top-down development process by repeated refinement of the system description. Since in practice this is hardly ever the case, support for the taking-back of erroneous design steps and the *revision of the system description* is indispensable for a practical tool. One simple revision concept might consist of the possibility to return to an earlier stage of the development process, substituting a previously taken erroneous design step by the correct one, and "replaying" the subsequently taken steps as far as possible, informing the user graphically about the inapplicability of "replayed" steps.

To guarantee the correctness of the development process using such a tool, all these described extensions must, of course, be proven correct in respect to the original method.

6 Concluding Remarks

We have described an approach to amalgamate a pragmatic software specification method taken out of daily industrial practice with a precise semantic background. The novel aspect of our approach is that we do not stop at a pure translation between informal and formal specifications but integrate both into a new development method that appears to the user as a structured graphical method. The formal background is then used to make the method much more elaborate in its methodical guidance. Moreover, the method offers an elegant way to ensure the consistency of a specification by construction. This seems to be superior to consistency tests as they have been defined in informal and formal methods up to now.

Of course, the described approach is not limited to the specific development method. The basic idea can be carried over to any other graphics-based method, including object-oriented methods. For example, [7] has exercised a quite similar approach for the complex method SSADM. The work that was presented here seems to be of a completely new kind compared to other activities in (formal or informal) system development. We have carried out here a piece of method development for a graphical method based on formal foundations. One can draw the conclusion that the analysis and improvement of practically used development methods can be a fruitful field of research (and also business!) for people with experience in both formal and informal development methods.

7 Acknowledgment

The authors would like to thank Alexander van der Vekens for contributions to an earlier version, and Konrad Slind for a careful reading of the article, as well as the reviewers for their helpful remarks.

8 References

[1] Broy, M., Dendorfer, C., Dederichs, F., Fuchs, M., Gritzner, T., Weber, R.: *The Design of Distributed Systems - An Introduction to FOCUS.* TUM-I9225, Technische Universität München, 1992.

[2] Bowen, J., Stavridou, V.: *The industrial take-up of formal methods in safety-critical and other areas: A perspective.* In: F. C. P. Woodcock, P. G. Larsen (eds), FME' 93, Lecture Notes in Computer Science Vol. 670, Springer 1993, pp. 183-195.

[3] Brauer, W.: *Automatentheorie.* Teubner, 1984.

[4] Facchi, C.: *Methodik zur formalen Spezifikation des IOS/OSI Schichtenmodells.* PhD-Thesis. Technische Universität München, 1995.

[5] Harel, D.: *Statecharts: a visual formalism for complex systems.* Science of Computer Programming **8** (1987) 231-274.

[6] Hoare, C.A.R.: *Mathematical models for Computer Science.* Working Material for Marktoberdorf Summer School 1994. Institut für Informatik, Technische Universität München, 1994.

[7] Hussmann, H.: *Formal Foundations for SSADM.* Habilitation Thesis, Technische Universität München, 1994.

[8] Hussmann, H.: *Indirect Use of Formal Methods in Software Engineering.* In: M. Wirsing (Ed): ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice, Seattle (WA), USA. Proceedings, April 1995, pp. 126-133.

[9] International Telecommunication Union: *Message Sequence Charts.* ITU-T Recommendation Z.120. Geneva, 1994.

[10] Lloyd, J.W.: *Foundations of Logic Prograamming.* Springer, 1984.

[11] Milner, R. *CCS - A Calculus for Communicating Systems.* Springer Lecture Notes in Computer Science 83, 1983.

[12] Semmens, L.T., France, R.B., Docker, T.W.G.: *Integrated structured analysis and formal specification techniques.* The Computer Journal **35** (1992) 600-610.