Data flow oriented test selection for LOTOS

HANS VAN DER SCHOOT & HASAN URAL
Department of Computer Science
University of Ottawa
Ottawa, Ontario K1N 6N5, Canada
{vdschoot,ural}@csi.uottawa.ca

Abstract

A data flow oriented test selection method for selectively generating abstract tests from a LOTOS specification is presented. This method is based on static data flow analysis to identify maximal IO-df-chains, which exhibit the associations between each specification output and those specification inputs that influence the output through definitions and uses of variables. The maximal IO-df-chains are then used to guide the derivation of test sequences. A new set of inference rules is introduced for this purpose. As an illustration, the test selection method is applied to the alternating bit protocol.

Keywords

specification languages, process algebras, LOTOS, modelling, testing, test selection

1 Introduction

Recently, a great deal of attention has been given to generating a set of tests from the specification of a distributed system formalized in SDL [1], ESTELLE [6], or LOTOS [3] for the purpose of testing an implementation of the system. In general, a system specification in SDL, ESTELLE, or LOTOS describes the externally observable behaviour of the system in terms of possible sequences of interactions between the system and its environment. Such a specification expresses the desired control and data flow aspects of the system that must be established by a possible system implementation as sequences of interaction identifiers and relationships between interaction parameters, respectively.

Clearly, both the control and data flow aspects of a possible implementation of a distributed system must be tested. Since exhaustive testing is generally infeasible, practical testing has been confined to using some test selection criteria to selectively construct a set of tests of manageable size from the given system specification. Although a wide variety of test selection criteria have been proposed for testing the control flow aspects of distributed systems [4, 8, 9, 13, 22, 23, 26, 27, 32], test selection criteria for testing the data flow aspects of such systems are just emerging

[7, 20, 28, 29, 30]. Usually, a test selection criterion is oriented towards testing either control flow or data flow aspects of a system. As a consequence, the tests constructed by the control flow oriented criteria are complementary to those constructed by data flow oriented criteria [11, 15, 19, 21, 31]. Therefore, both types of test selection criteria are needed for the construction of a comprehensive set of complementary tests.

In this paper, a method for the selective generation of data flow oriented tests from a LOTOS specification is developed [16]. The method involves the application of the static data flow analysis technique [10]. First, the given specification is represented as a flowgraph that exhibits the associations between definitions and uses of each variable employed in the specification. Using this information, the effects of values accepted by the specified system (i.e. specification inputs) on values offered to its environment (i.e. specification outputs) are identified. These input-output relations form the basis for a data flow oriented test selection criterion, called the *maximal IO-df-chains criterion*. Then, the maximal IO-df-chains in the flowgraph of the specification are identified and used as guides by inference rules to derive abstract tests. For this purpose, the concept of *guided inference rules* for LOTOS is introduced. These inference rules provide a means of constructing test sequences based on the maximal IO-df-chains. The resulting set of tests provides the capability of determining whether an implementation establishes the desired flow of data expressed in the given specification, and thus complements the tests constructed by control flow oriented test selection criteria.

The rest of the paper is organized as follows. In Section 2 we define the maximal IO-df-chains criterion. In Section 3 we show how the maximal IO-df-chains are transformed into guides which are used by inference rules to selectively generate test sequences. As an illustration, in Section 4 we apply the test selection method to the alternating bit protocol. Finally, we give concluding remarks in Section 5.

2 Maximal IO-df-chains Criterion

2.1 Terminology

The terms used throughout this section to facilitate the precise description of the maximal IO-df-chains criterion are grouped into control flow related concepts and data flow related concepts.

2.1.1 Control Flow Related Concepts

For ease of presenting the proposed test selection criterion, a behaviour expression in LOTOS is represented by a digraph $G(N, E)$, called *flowgraph*, where N is the set of nodes corresponding to syntactic elements in the behaviour expression and E is the set of directed edges indicating

possible flow of control between the nodes in N (see Fig. 1). The flowgraph of a behaviour expression can be seen as the abstract syntax tree [16] of the expression extended with the representation of dynamic semantic aspects such as enabling and the value passing type of interactions. That is, in addition to the syntactic structure of a behaviour expression B, the flowgraph of B models the potential flow of control that may be caused by enabling and by the value passing type of interactions between subexpressions in B.

The following types of nodes and edges are distinguished in the flowgraph $G(N, E)$ of B:

- an o-node (e.g. the circles in Fig. 1) representing an observable action in B;
- an i-node (e.g. the double circles in Fig. 1) representing an internal action in B;
- a p-node (e.g. the rectangles in Fig. 1) representing a process instantiation in B;
- a c-node (e.g. the diamonds in Fig. 1) representing a nullary, unary (except for the action prefix), or binary operator in B;
- an s-node (e.g. the patterned circle in Fig. 1) representing a potential value passing type of interaction in B;
- an r-edge (e.g. the plain edges in Fig. 1) representing static control flow in B;
- an e-edge (e.g. the bold edge in Fig. 1) representing dynamic control flow caused by enabling in B;
- an s-edge (e.g. the dashed edges in Fig. 1) representing dynamic control flow caused by a potential value passing type of interaction in B;

All the nodes in N except s-nodes are assigned appropriate labels. For instance, a c-node is labelled with the operator being represented, whereas an o-node or i-node is labelled with the action that is represented (see Fig. 1).

The classification of E into edges representing static control flow (i.e. r(egular)-edges) and edges representing dynamic control flow (i.e. e(nable)-edges or s(ynchronization)-edges) is merely based on the syntactic structure of B. The r-edges are the edges as present in the abstract syntax tree of B, whereas the e-edges and s-edges reflect dynamic aspects of B. The precise need for this classification becomes clear in the second phase of our test selection method, which is discussed in Section 3.

The mapping from a LOTOS specification into its flowgraph is defined by recursion over LOTOS behaviour expressions, i.e. the flowgraph of a compound behaviour can be constructed on the basis of the flowgraphs of its components. The definition assumes behaviour expressions to be free from "par" constructs, and "choice" constructs over gates. Furthermore, the definition requires *recursive* process instantiations to be tail recursive without gate relabelling. Although these assumptions restrict the applicability of the flowgraph mapping, they do not necessarily limit the scope of our test selection method for LOTOS. First of all, one can always fall back on

dynamic expansion to obtain a recursive process instantiation without gate relabelling. The corresponding process definition can be unfolded by repeated process instantiations until the actual gates are found to be identical to the formal gates. As a list of a finite number of gates only has a finite number of permutations, the expansion will terminate after a finite number of steps. Secondly, the assumption that recursive process instantiations should be tail recursive is due to the common problem of representing non-tail recursion in a graph. However, this kind of recursion can always be avoided in the attempted definition of a process behaviour. Finally, the "par" construct and the "choice" construct over gates are only short-hands of expressions that are handled by the flowgraph mapping.

We only give a brief sketch of the flowgraph mapping. A precise definition of the mapping is given in [24, 25]. Each operator, except the action prefix operator, is represented by a *c*-node with the corresponding operator label. In case of an action prefix expression g; $B[P]$ an *o*-node or *i*-node is constructed, depending on whether $g \neq \mathbf{i}$ or $g = \mathbf{i}$, respectively. This node has an outgoing *r*-edge to the flowgraph of B, to which the possible selection predicate $[P]$ is attached. The expressions **hide** D **in** B and $[P] \rightarrow B$ are represented likewise, with their respective *c*-nodes having an outgoing *r*-edge to the flowgraph of the argument expression B. In the latter case the guard $[P]$ is associated with this edge. Furthermore, the *o*-nodes in the flowgraph of the hide expression corresponding to the hidden actions in D are replaced by *i*-nodes. The *c*-nodes representing a binary operator have two outgoing *r*-edges, one to the flowgraph of the left argument expression and one to the flowgraph of the right argument expression. In addition, *s*-nodes and *s*-edges are constructed between all potentially synchronizing actions (or nodes) in the flowgraph of a parallel composition expression if the effect of such potential synchronization is value passing (see Fig. 1.a). In case of an enable expression *e*-edges are added between the appropriate nodes (see Fig. 1.b). Finally, a process instantiation is represented by a *p*-node with the process name as its label. An outgoing *r*-edge is constructed to the flowgraph of the expression within the corresponding process definition. This leads to recursion in the flowgraph if the process instantiation is recursive (i.e. tail recursive without gate relabelling). The *o*-nodes and *i*-nodes are then subjected to relabelling, which is well defined due to the above restriction on recursive process instantiations. The various construction rules are illustrated in the figures throughout this paper, in particular in the Appendix.

A ***path*** $(n_1, n_2, \ldots, n_m)$ in a flowgraph $G(N, E)$ is a finite sequence of two or more nodes in N, such that for all i, $1 \leq i \leq m-1$, $(n_i, n_{i+1}) \in E$ $(m \geq 2)$. A path $(i_1, \ldots, i_k)$ is a ***subpath*** of a path $(n_1, \ldots, n_m)$ if there exists a δ, $0 \leq \delta \leq m-k$, such that for all j, $1 \leq j \leq k$, $i_j = n_{j+\delta}$. A ***cycle*** is a path in which the first and the last node are identical. A *path* P_1 *is* ***covered*** *by a path* P_2 if P_1 is a subpath of P_2. A ***regular path expression*** (***RPE***) is inductively defined as follows:

- an empty string, a node or a path is an RPE;

- P^* is an RPE, denoting zero or more chained occurrences of an RPE P;
- $P_1 \mid P_2$ is an RPE, denoting a choice between RPEs P_1 and P_2;
- P_1, P_2 is an RPE, denoting sequencing of RPEs P_1 and P_2.

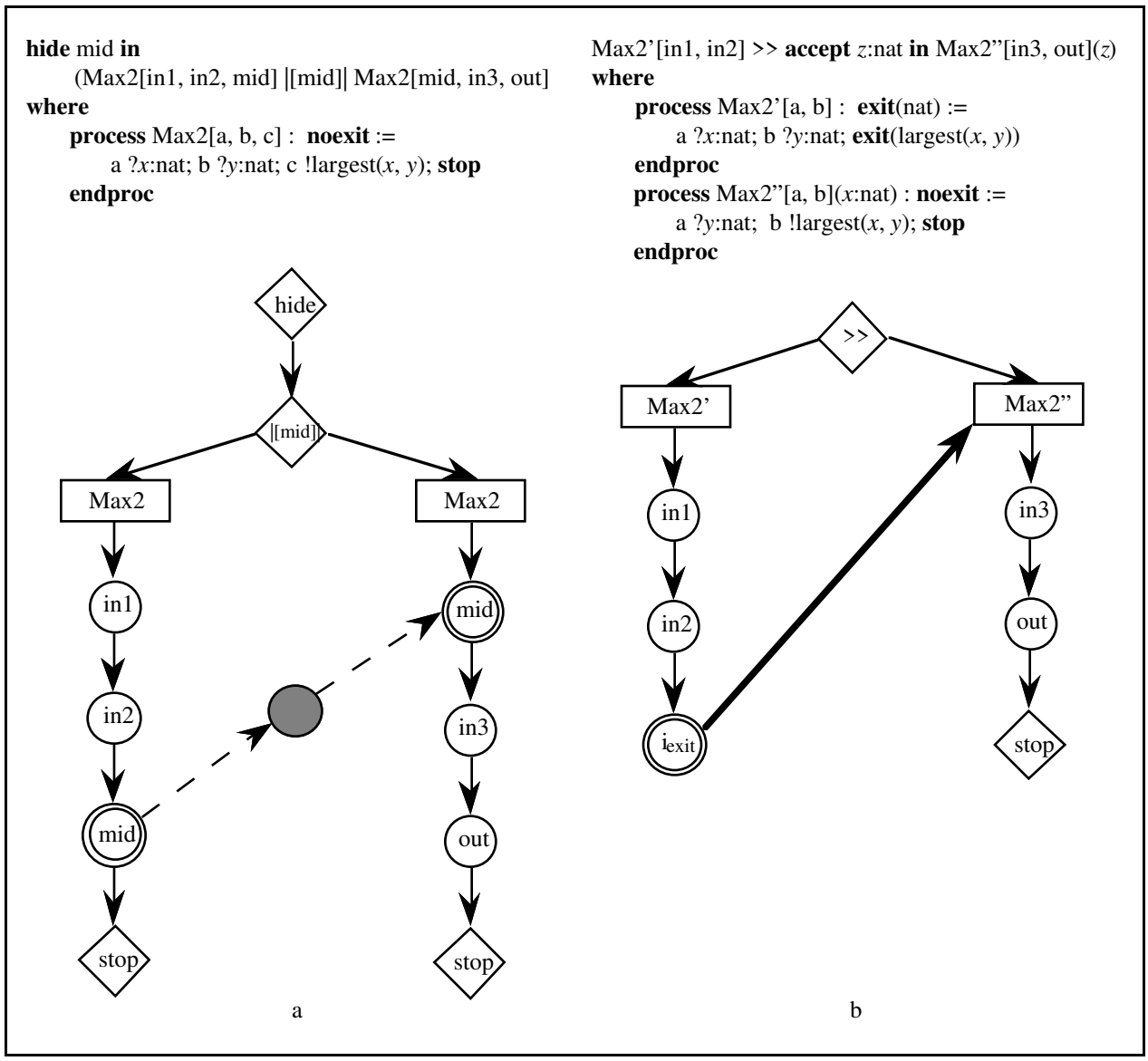

Fig. 1 – Modelling a behaviour expression as a flowgraph

2.1.2 Data Flow Related Concepts

Variable occurrences in a LOTOS behaviour expression, occurring either as part of a syntactic element associated with a node in a flowgraph, or in a guard or selection predicate associated

with an edge in the flowgraph, are classified as definitions or uses (see Fig. 2[1]). A ***definition*** (or ***def***) of a variable x at node n (denoted by d_n^x) is an occurrence of x by which it takes a value, and a ***use*** of a variable x at node n (denoted by u_n^x) or on edge (n, m) (denoted by $u_{(n, m)}^x$) is an occurrence of x by which its value is referenced. In LOTOS, a def or a use of a variable can then be interpreted as a variable occurrence in an "identifier-declaration" or a "value-expression", respectively [16].

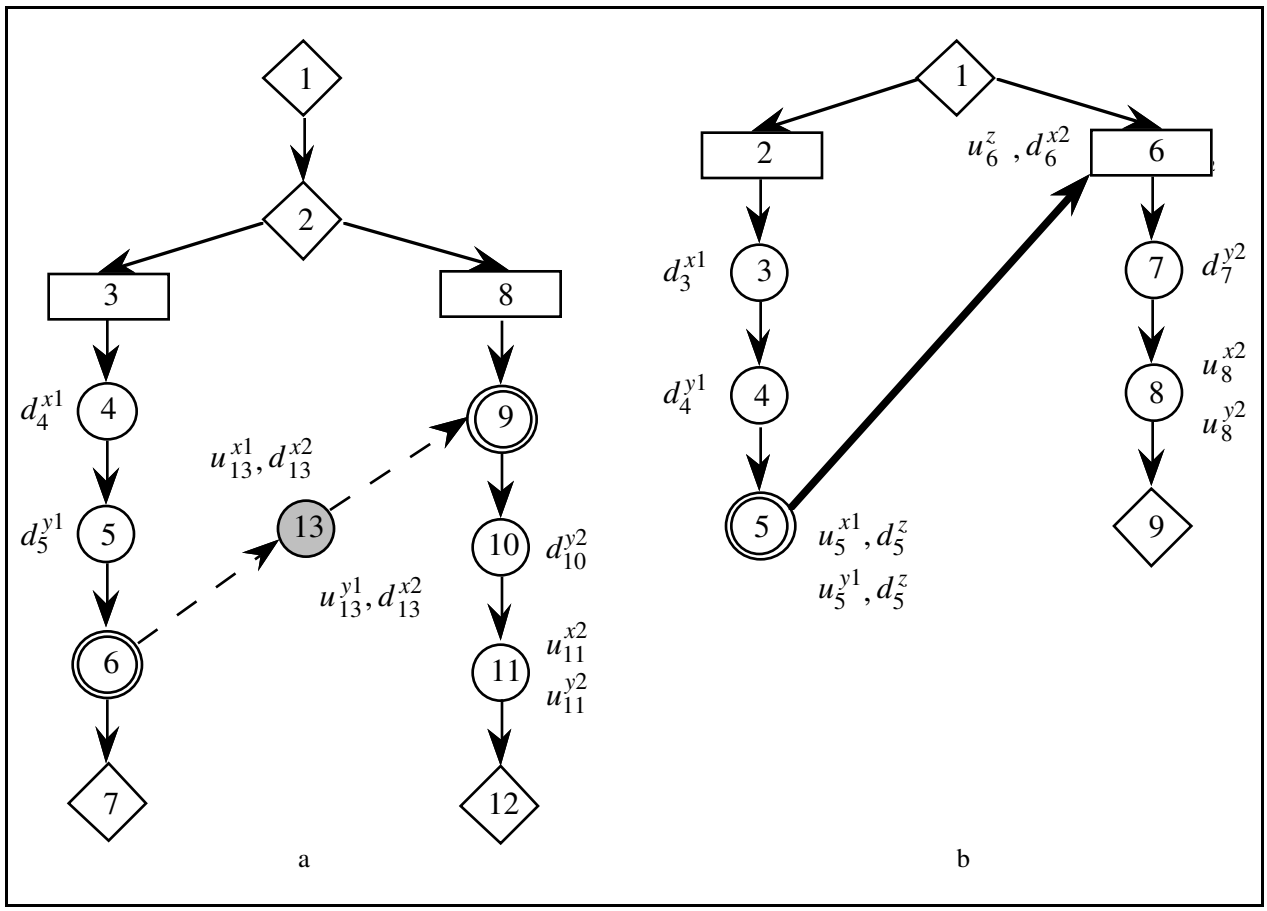

Fig. 2 – Classifying variables in a behaviour expression

The operational semantics of LOTOS allows a variable in a behaviour expression to take the value of a particular value expression in a behaviour expression. For instance, a process may accept a value offered by another process by means of a rendez-vous or by means of enabling. In such cases of variable instantiation, the def of a variable, say x, ***is caused by*** uses of all the variables, say $y_1, \ldots, y_r$, in the value expression, or equivalently, x is defined as a function of $y_1, \ldots, y_r$. This is denoted by $u_n^{y_i}, d_n^x$ $(1 \le i \le r)$. Note that this notation implies that a particular

[1] The variables are renamed in order to avoid global conflicts in the flowgraph.

def and the corresponding causing uses are all associated with a single node (for instance, an *s*-node in case of value passing, see Fig. 2).

The concept of uses causing defs also appears in the case of process instantiations; a formal parameter x_i of a process definition may be instantiated with an actual parameter E_i of a corresponding process instantiation. For *recursive* process instantiations we make the distinction between $E_i \neq x_i$ and $E_i = x_i$. In the first case, we indeed record that a def of x_i is caused by each variable use in E_i, as explained above. However, in the latter case no such association between the def and the use of x_i is recorded since the value of x_i is not modified. The rationale here is that no update is performed on x_i. Fig. 3 illustrates the idea. Node 6 represents the recursive instantiation `P[a](succ(x))` and contains u_6^x, d_6^x since the formal parameter x is instantiated with the new value succ(x). Node 4 does not contain a def or use since the actual parameter x in the recursive instantiation `P[a](x)` is syntactically identical to the formal parameter in the header of the process definition.

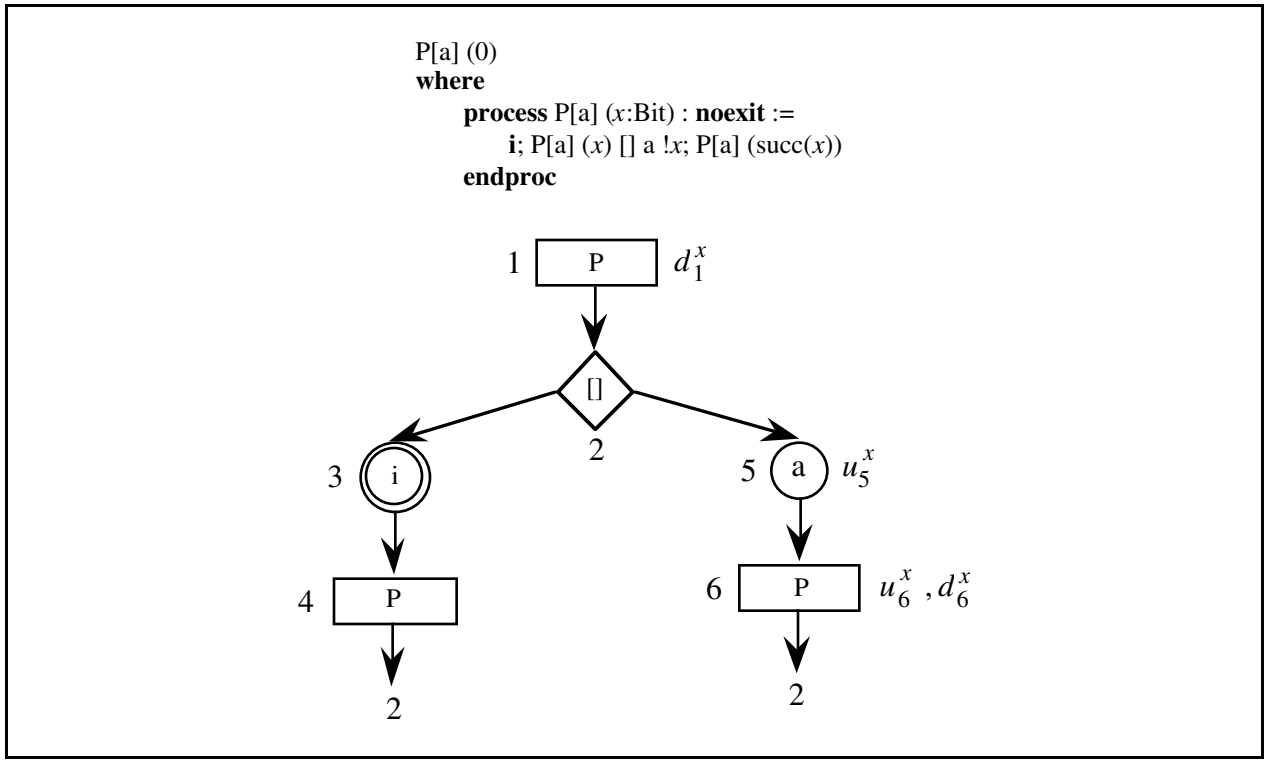

Fig. 3 – Recursion in a flowgraph

It must be noted that although we do not make a distinction in this paper between uses occurring in a node (i.e. computational uses) and on an edge (i.e. predicate uses), such a distinction facilitates the application of existing data flow oriented test selection criteria such as

the ones given in [11]. A path (n_1, n_2, … , n_{m-1}, n_m) is a ***def-clear path with respect to (wrt) a variable x*** from node n_1 to node n_m, or from node n_1 to edge (n_{m-1}, n_m), if either m = 2, or m > 2 and there are no defs of x at nodes n_2 to n_{m-1}. For ease of reference, $q$denotes either a node or an edge. A def d_i^x and a use u_q^x form a ***du-pair*** (d_i^x, u_q^x) if there is a def-clear path wrt x from node i to q.

2.2 The Proposed Test Selection Criterion

Before defining the maximal IO-df-chains criterion, the motivation behind the criterion is detailed. First of all, as advocated for procedural program testing [15, 18], we want to link du-pairs in a behaviour expression to form a sequence of du-pairs. It allows us to trace a sequence of computations which capture particular control and data dependencies in the behaviour expression. This gives rise to the notion of *affect* [29, 31]. A def of a variable x ***affects*** a use of a variable y if either

a) the def of x and the use of y form a du-pair (i.e. x and y are the same variable), or
b) a def of a variable z is caused by a use of x that forms a du-pair with the def of x, and the def of z affects the use of y .

For example, u_8^{x2} in Fig. 2.b is affected by d_6^{x2} (directly) and by d_3^{x1} and d_4^{y1} (indirectly). A ***df-chain*** is an ordered sequence ($d_{n_1}^{x_1}$, $u_{n_2}^{x_1}$, … , $d_{n_m}^{x_m}$, $u_{n_{m+1}}^{x_m}$) of du-pairs ($d_{n_1}^{x_1}$, $u_{n_2}^{x_1}$), … , ($d_{n_m}^{x_m}$, $u_{n_{m+1}}^{x_m}$), such that $u_{n_{i+1}}^{x_i}$, …, $u_{n_{m+1}}^{x_m}$ are affected by $d_{n_i}^{x_i}$. Examples of df-chains in Fig. 2.a are (d_4^{x1}, u_{13}^{x1}, d_{13}^{x2}, u_{11}^{x2}), (d_5^{y1}, u_{13}^{y1}, d_{13}^{x2}, u_{11}^{x2}) and (d_{10}^{y2}, u_{11}^{y2}).

The particular df-chains of interest are those that start with “inputs” and terminate with “outputs”. The motivation here is to identify the intended functionality of a system in terms of the effects of values accepted from the system's environment on values offered to the environment. An ***input*** of a specification is either

a) a def of a variable at an *o*-node (e.g. d_4^{x1} in Fig. 2.a), or
b) a def of a variable at a *p*-node which is not caused by other variables.

According to this definition, the variables referred to in identifier declarations of an experiment offer list (i.e. “?*x*:*s*”) associated with an observable gate, which do not obtain a value by means of synchronization with another (specified) process (such defs would be associated with an *s*-node rather than an *o*-node), are considered as inputs. Also, variable initializations through process instantiations, initialized by value expressions consisting of only ground terms, are considered as inputs. An ***output*** of a specification is

a) a use of a variable at an *o*-node (e.g. u_{11}^{x2} in Fig. 2.a), or
b) a use of a variable at an *s*-node with an incoming *s*-edge from and an outgoing *s*-edge to an

o-node, or

c) a use of a variable at a *c*-node with label "exit".

Accordingly, the variables referred to in value expressions of an experiment offer list (i.e. "!*E*") associated with an observable gate are considered as outputs. Also, variables occurring in an "exit" construct without an enabling context are considered as outputs[2]

We observe that capturing the intended system functionality in terms of the effects of inputs on outputs can be accomplished by input-output relation analysis. Let I and O be the sets of all inputs and outputs in a specification, respectively. A binary relation w.r.t. the notion of affect, the ***input-output relation***, is defined as $\Re_{IO} = \{(i, o) \mid i \in I, o \in O$, and i affects $o\}$. A df-chain $(d_{n_1}{}^{x_1}, u_{n_2}{}^{x_1}, \ldots, d_{n_m}{}^{x_m}, u_{n_{m+1}}{}^{x_m})$ is an ***IO-df-chain*** if $(d_{n_1}{}^{x_1}, u_{n_{m+1}}{}^{x_m}) \in \Re_{IO}$ and there are at most *two* occurrences of any $u_{n_i}{}^{x_{i-1}}, d_{n_i}{}^{x_i}$ $(2 \le i \le m)$. The latter condition is both necessary and sufficient to cover all du-pairs in a LOTOS specification and to capture all associations between inputs and outputs in the specification. This is demonstrated by the scenario in Fig. 3 which shows recursion characteristic for the flowgraph of a LOTOS specification. In order to cover the du-pair $(d_6{}^x, u_6{}^x)$ and to capture the association between the input $d_1{}^x$ and the output $u_5{}^x$, a df-chain that starts with $d_1{}^x$ and terminates with $u_5{}^x$ requires two occurrences of $u_6{}^x, d_6{}^x$ yielding IO-df-chain $(d_1{}^x, u_6{}^x, d_6{}^x, u_6{}^x, d_6{}^x, u_5{}^x)$. From the above definitions it follows that

a) $\Re_{IO}$ is a finite set since $\Re_{IO} \subseteq I \times O$ and for any specification I and O are finite sets, and

b) there exists at least one IO-df-chain for every $(i, o) \in \Re_{IO}$.

As an example, the input-output relation in the flowgraph of Fig. 2.a is $\Re_{IO} = \{(d_{10}{}^{y2}, u_{11}{}^{y2}), (d_4{}^{x1}, u_{11}{}^{x2}), (d_5{}^{y1}, u_{11}{}^{x2})\}$ which is justified by the IO-df-chains $(d_4{}^{x1}, u_{13}{}^{x1}, d_{13}{}^{x2}, u_{11}{}^{x2})$, $(d_5{}^{y1}, u_{13}{}^{y1}, d_{13}{}^{x2}, u_{11}{}^{x2})$ and $(d_{10}{}^{y2}, u_{11}{}^{y2})$, respectively.

The objective of the proposed criterion is, for every $(i, o) \in \Re_{IO}$, to identify and subsequently cover all the IO-df-chains through which i affects o. In general, the identification of IO-df-chains may yield several IO-df-chains which are prefixes of other IO-df-chains. For example, in the special case of an IO-df-chain which is also a du-pair, it is likely that this du-pair is linked with other du-pairs to form the prefix of a longer IO-df-chain. But, for the data flow coverage of a specification in terms of effects of inputs on outputs, it is sufficient to consider a minimal set of *maximal* IO-df-chains, i.e. IO-df-chains which are not prefixes of any other, distinct IO-df-chain. Formally, an IO-df-chain $(d_{i_1}{}^{y_1}, u_{i_2}{}^{y_1}, \ldots, d_{i_k}{}^{y_k}, u_{i_{k+1}}{}^{y_k})$ $(k \ge 1)$ is a ***maximal IO-df-chain*** if there is no IO-df-chain $(d_{n_1}{}^{x_1}, u_{n_2}{}^{x_1}, \ldots, d_{n_m}{}^{x_m}, u_{n_{m+1}}{}^{x_m})$ $(m > k)$ such that for all j, $1 \le j \le k+1$, $i_j = n_j$ and $y_j = x_j$.

[2] A *c*-node with label "exit" represents an "exit" construct in the absence of an enabling context. An "exit" construct in the presence of an enabling context is represented by an *i*-node with label "i_{exit}", which has an outgoing *e*-edge to the flowgraph of the enabled behaviour.

Let P be an RPE. The ***activating path expression*** (***APE***) ***for a du-pair*** (d_i^x, u_q^x) is an RPE (i, P, q) that enumerates all paths which are def-clear wrt x from i to q. The ***APE for a df-chain*** ($d_{n_1}^{x_1}$, $u_{n_2}^{x_1}$, … , $d_{n_m}^{x_m}$, $u_{n_{m+1}}^{x_m}$) is an RPE (n_1, P_1, n_2, P_2, n_3, … , n_m, P_m, n_{m+1}), where (n_i, P_i, n_{i+1}) is the APE for the du-pair ($d_{n_i}^{x_i}$, $u_{n_{i+1}}^{x_i}$) ($1 \le i \le m$). Each path enumerated by an APE for a df-chain is an ***activating path*** for the df-chain. Thus, an APE for a maximal IO-df-chain enumerates all the activating paths for the maximal IO-df-chain. An ***IO-df-chain is covered by a path*** if an activating path for the IO-df-chain is a subpath of the path.

The proposed data flow oriented test selection criterion is now defined in the following manner. The ***maximal IO-df-chains criterion*** requires that each maximal IO-df-chain in a flowgraph is covered by a path at least once.

3 Selective Test Generation

Clearly, satisfying the maximal IO-df-chains criterion can be achieved by selecting a minimal set of test paths to cover an activating path for each maximal IO-df-chain. However, not all activating paths enumerated by the APE for a maximal IO-df-chain may be *feasible* (i.e. executable). A path is *infeasible* if a predicate on the path can never be satisfied due to the existence of a logical contradiction or an assignment on that path which makes the predicate always false. For instance, consider the path (62, 63, 64, 65) in the flowgraph of the ABP specification (see Annex B, Fig. B.1 and Table B.1-B.4) which corresponds to line numbers 116, 104 and 105 in the ABP specification in Annex A. On this path the guard [*ok*] can never be satisfied as the variable *ok* is bound to false. Unfortunately, it can be shown from decidability theory that selection of infeasible paths cannot be uniformly avoided by using only the static information contained in the given specification. Moreover, there may not be a feasible activating path at all among those enumerated by the APE for a maximal IO-df-chain, in which case the maximal IO-df-chain is called an *infeasible maximal IO-df-chain*.

In order to generate feasible tests based on the maximal IO-df-chains criterion, feasible activating paths must be selected for feasible maximal IO-df-chains. It is important to note that the activating paths enumerated by the APE for a maximal IO-df-chain in the flowgraph of a behaviour expression correspond to particular traces of actions in the behaviour expression. These traces of actions can be obtained by extracting the labels of the *o*-nodes and *i*-nodes in the corresponding paths. Clearly, in order to determine the feasibility of such traces and thus the feasibility of the corresponding activating paths, the dynamic semantics of LOTOS has to be taken into account. This semantics is defined in terms of axioms and inference rules [16].

Recently, a new type of inference rules has been proposed [14]. These so-called *goal oriented inference rules* are capable of generating traces of actions leading to a preselected action

in a behaviour expression. The inference rules are guided by the static derivation path of a particular action. The *static derivation path* (SDP) of an action is a path which statically locates the action in the abstract syntax tree of a behaviour expression [14]. Although the flowgraph of a behaviour expression is not an abstract syntax tree, it represents the syntactic structure of the behaviour expression similar to the abstract syntax tree of the expression. Therefore, SDPs also allow one to locate statically particular actions in the flowgraph of the behaviour expression. Moreover, any node and thus the defs and uses of variables associated with nodes (or edges) are located in the flowgraph by SDPs. This implies that the concept of goal oriented inference rules provides a capability to derive feasible tests based on the selected test paths.

By utilizing an adaptation of the goal oriented inference rules, we develop an approach for the generation of tests covering the set of feasible maximal IO-df-chains of a LOTOS specification. The outline of this approach is as follows. First, each maximal IO-df-chain is mapped into a set of SDPs. This is done by subsequently splitting the APE for the maximal IO-df-chain into a set of RPEs, extending each RPE to a rooted path expression, and generating a set of SDPs for each rooted path expression. These SDPs are then merged to form a guide for the derivation of the test sequence which covers the maximal IO-df-chain. Merging SDPs is achieved by transforming each SDP into an augmented SDP (ASDP) and by unifying the resulting ASDPs [17].

3.1 Static Derivation Paths

As introduced in [14], the SDP of a particular action occurrence g in a behaviour expression B is a sequence identifying a (rooted) path in the abstract syntax tree of B that leads to g. This path reflects the directed traversal of the operators composing B (e.g. left part of choice, then action prefix, and then right part of disable, etc.). An SDP has the following form:

- [] stands for the empty SDP;
- [$e.s$] stands for an SDP containing the element e followed by the SDP s.
- [e_1, … , e_n] stands for [e_1.[…[e_n.[]] …]].

An element of an SDP is a symbol indicating a type of behaviour construct in the abstract syntax tree. For example, “pf” and “inst” are the symbols indicating the action prefix operator and the process instantiation, respectively. In addition, branches for the left and right behaviour of a binary operator (i.e. |[]|, [], [>, and >>) are indicated by “^l” and “^r”, respectively. For example, “par^l” indicates the left part of a parallel operator.

The concept of SDP is solely based on the syntax of LOTOS. As the flowgraph representation preserves this syntax, the SDP of an action in a behaviour expression B identifies a path in the flowgraph $G(B)$ which starts at the entry node of $G(B)$ (*rooted* path) and terminates with the node

representing the action. The path does not traverse any *s*-edges or *e*-edges which represent dynamic semantic matters rather than syntactic matters. In general, each rooted path in a flowgraph not traversing *s*-edges or *e*-edges is identified by a particular SDP. The elements of this SDP follow directly from the syntactic LOTOS elements represented by the various types of nodes and node labels in the path (see algorithm "SDP").

```
Specification Gambling_machine[in, out] :=
hide mid in
Input[in, mid] |[mid]| Output[mid, out]
where
    process Input[a, b] : noexit :=
        a ?x:Coin; b !x ; Input[a, b]
    endproc
    process Output[a, b] : noexit :=
        a ?y:Coin;  (i; Output[a, b] [] b !(2*y); Output[a, b]
    endproc
endspec
```

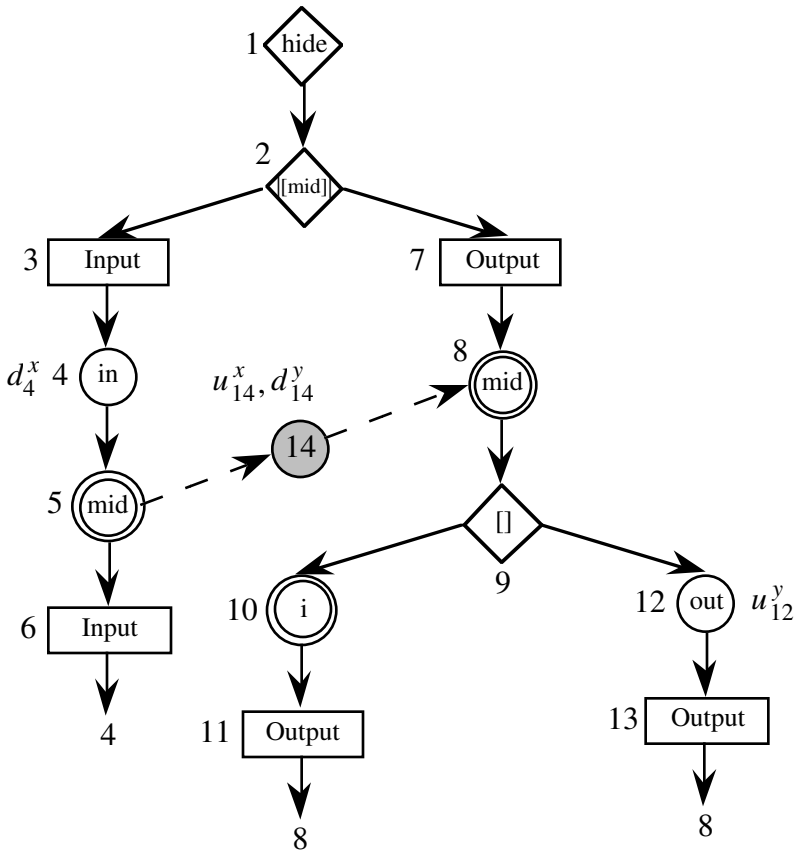

Fig. 4 – A gambling machine

3.1.1 Mapping Maximal IO-df-chains to Static Derivation Paths

Mapping a maximal IO-df-chain to a set of SDPs is performed in two steps. In the first step, the *s*-nodes, *s*-edges, and *e*-edges in the APE for the maximal IO-df-chain are eliminated. This is done by splitting the APE at the appropriate places, which results in a number of RPEs. In the second step, each of these RPEs is extended to a rooted path expression. Each rooted path expression enumerates all rooted paths that terminate with the RPE concerned. The SDPs matching the rooted path expression are then generated according to the algorithm "SDP".

As an example, consider the specification in Fig. 4. This specification describes a gambling machine which either keeps or pays twice the amount of money put in. The only maximal IO-df-chain for this specification is (d_4^x, u_{14}^x, d_{14}^y, u_{12}^y). The APE for this maximal IO-df-chain is

(4, 5, 14, ((8, 9, 10, 11)* | (8, 9, 12, 13)*)*, 8, 9, 12)

Splitting the APE at *s*-node 14 yields {(4, 5), (((8, 9, 10, 11)* | (8, 9, 12, 13)*)*, 8, 9, 12)}. The two rooted path expressions terminating with these RPEs are given by (1, 2, 3, (4, 5, 6)*, 4, 5) and (1, 2, 7, ((8, 9, 10, 11)* | (8, 9, 12, 13)*)*, 8, 9, 12), respectively. Finally, the matching SDPs are

[hd, par^l, inst, (pf, pf, inst)*, pf, pf]
[hd, par^r, inst, pf, ((ch^l, pf, inst, pf)* | (ch^r, pf, inst, pf)*)*, ch^r, pf]

3.1.2 Merging Static Derivation Paths

If one considers a set of SDPs for a maximal IO-df-chain, it can be observed that these SDPs are generally related in the sense that 1) one SDP reflects the left traversal of a particular occurrence of a binary operator, whereas another SDP reflects the right traversal of the operator, or 2) one SDP is an extension of another SDP. The SDPs given above for the flowgraph in Fig. 4 show an example of the first case. The second case may occur when the APE for a maximal IO-df-chain contains two *s*-nodes between flowgraphs of behaviour expressions B_1 and B_2, where one of the *s*-nodes represents value passing from B_1 to B_2 and the other from B_2 to B_1.

The relations between SDPs are established by merging them. First, each SDP is transformed into a new structural unit, called *augmented* SDP (ASDP), and the resulting ASDPs are then subjected to unification [17]. An ASDP is defined using Backus-Naur Form as follows:

<ASDP> ::= _ | [] | [<OP_1>.<ASDP>] | [<OP_2>(<ASDP>, <ASDP>)]
<OP_1> ::= ex | pf | hd | ch^l | ch^r | inst | gd | let | sum
<OP_2> ::= en | dis | par

To facilitate the use of [e_1, … , e_n] as an equivalent notation for [e_1.[e_2.[…[e_n.[]] …]]], an operator "∘" is further used to denote the concatenation of two ASDPs.

```
Algorithm SDP:
α, β, γ, and κ denote the set of o-nodes, i-nodes, p-nodes, and c-nodes in the flow-
graph of a LOTOS specification, respectively. Furthermore, label(n) denotes the label
of node n, and left_child(n) denotes the left successor node of n.

Input: a path (n_1, ..., n_m) not traversing s-edges or e-edges
Output: an SDP

sdp := [];
i := m;
while i ≥ 1 do
    (* an o-node represents an observable action; the symbol "pf" is prefixed to the
    current SDP indicating an application of the action prefix operator*)
    case (n_i ∈ α) do sdp := [pf.sdp] endcase
    (* an i-node represents an unobservable action resulting either implicitly from the
    enable operator or explicitly from the action prefix operator; depending on the
    node's label the symbol "ex" or "pf" is prefixed to the current SDP *)
    case (n_i ∈ β) do
        if label(n_i) = i_exit
        then sdp := [ex.sdp]
        else sdp := [pf.sdp]
        endif
    endcase
    (* a p-node represents a process instantiation; the symbol "inst" is prefixed to
        the current SDP indicating an application of process instantiation *)
    case (n_i ∈ γ) do sdp := [inst.sdp] endcase
    (* a c-node represents an operator other than the action prefix operator; the
appropriate symbol indicating an application of the operator is prefixed to the
current SDP; in case of a binary operator the symbol is suffixed by "^l" or "^r"
indicating the direction of application *)
    case (n_i ∈ κ) do
        case label(n_i) = exit do sdp := [ex.sdp] endcase
        case label(n_i) = hide do sdp := [hd.sdp] endcase
        case label(n_i) = guard do sdp := [gd.sdp] endcase
        case label(n_i) = sum do sdp := [sum.sdp] endcase
        case label(n_i) = let do sdp := [let.sdp] endcase
        case label(n_i) ∈ {|[]|, [], >>, [>} do
            case label(n_i) = |[]| do binop := par endcase
            case label(n_i) = [] do binop := ch endcase
            case label(n_i) = >> do binop := en endcase
            case label(n_i) = [> do binop := dis endcase
            if left_child(n_i) = n_i+1
            then sdp := [binop^l.sdp]
            else sdp := [binop^r.sdp]
            endif
    endcase
endcase
i := i-1
endwhile
```

An ASDP differs from an SDP in the following respects:

- an ASDP may be unspecified, denoted as "_", indicating that no particular ASDP is required;
- branches for the left and the right behaviour of the enable, disable, and parallel operators are

indicated implicitly by a left and a right argument, rather than explicitly by "^l" and "^r".

By giving two arguments to the elements representing the enable, disable, and parallel operator, the left and the right traversal of such an operator can be considered simultaneously. Note that there is no need to do so for the choice operator, as the semantics of this operator makes its left and right behaviour mutually exclusive.

Formally, the transformation of an SDP into an ASDP is defined by the function μ:SDP $\rightarrow$ ASDP as follows (*s* stands for an SDP):

$\mu([]) = _$
$\mu([<OP_1>.s]) = [<OP_1>.\mu(s)]$
$\mu([<OP_2>\hat{}l.s]) = [<OP_2>(\mu(s), _)]$
$\mu([<OP_2>\hat{}r.s]) = [<OP_2>(_, \mu(s))]$

The empty SDP "[]" is mapped to the unspecified ASDP "_" for the purpose of unification. An ASDP which consists entirely of elements "$<OP_1>$" may correspond to (a part of) an SDP that is contained in another SDP, where both SDPs result from a single path in the flow graph (see the discussion in the first paragraph of this subsection). This ASDP should therefore always allow a possible extension; if the corresponding SDP is extended by another SDP, their matching ASDPs still unify. When the unification is done these particular unspecified ASDPs can simply be replaced by the empty ASDP.

The application of the transformation function μ to the SDPs corresponding to the maximal IO-df-chain for the specification in Fig. 4 results in the following ASDPs:

[hd, par([inst, (pf, pf, inst)*, pf, pf]$\circ$_, _)]
[hd, par(_, [inst, pf, ((ch^l, pf, inst, pf)* | (ch^r, pf, inst, pf)*)*, ch^r, pf]$\circ$_)]

Unifying these ASDPs yields

[hd, par([inst, (pf, pf,inst)*, pf, pf], [inst,pf, ((ch^l, pf, inst, pf)* | (ch^r, pf, inst, pf)*)*, ch^r,pf])]

As explained in the next subsection, the ASDPs enumerated by this expression will be used as guides in the application of the inference rules for the derivation of a feasible test sequence which covers the maximal IO-df-chain (d_4^x, u_{14}^x, d_{14}^y, u_{12}^y).

3.2 Guided Trace Derivation

Before we address the use of ASDPs as guides for the generation of tests, we review the required terminology.

3.2.1 Definitions

Traces and Trace Operators

In [12], a trace is defined as a finite sequence of symbols which are denoted by identifiers and may represent observable actions (i.e. gates). Each trace can then be interpreted as a sequence of interactions that may take place between a process and its environment. Although this definition is suitable in the context of Basic LOTOS, it needs to be modified to deal with Full LOTOS. We consider a trace as a finite sequence of *structured* actions $gd_1 \ldots d_n$, where d_i $(1 \leq i \leq n)$ is an experiment offer instance. For convenience, "action" is used instead of "structured action" when the interpretation is clear from the context. A trace is denoted as follows:

- $\langle\rangle$ stands for the empty trace;
- $\langle a.t \rangle$ stands for a trace containing the action a followed by the trace t.

Similar as for (A)SDPs, we use $\langle a_1, \ldots, a_n \rangle$ and $\langle a_1.\langle a_2.\langle \ldots \langle a_n.\langle\rangle \ldots \rangle\rangle\rangle$ as equivalent notations.

Many operators and composition functions have been defined on traces [12]. The ones needed in this paper have been adapted for Full LOTOS and are given below:

- the *projection* of a trace t on an alphabet A, denoted by $t \lfloor A$, is the trace t excluding all actions whose gate names are not in A;
- the *complementary projection* of a trace t on an alphabet A, denoted by $t \rceil A$, is the trace t excluding all actions whose gate names are in A;
- the *concatenation* of two traces t_1 and t_2, denoted by $t_1 \circ t_2$, is the trace consisting of the sequence of actions of t_1 followed by the sequence of actions of t_2;
- the *renaming* of a trace t, denoted by $t[g_1 / h_1, \ldots, g_n / h_n]$, is the trace t in which each h_i is replaced by g_i $(1 \leq i \leq n)$;
- the *last element* of a trace t, denoted by $t^\wedge$, is the last action in a non-empty trace t;
- the *merging* of two traces $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$ *with respect to a synchronization alphabet* S, denoted by $\langle a_1, \ldots, a_n \rangle |S| \langle b_1, \ldots, b_n \rangle$, is a trace of the behaviour expression $a_1; \ldots; a_n;$ **stop** $|[S]|$ $b_1; \ldots; b_n;$ **stop**.

Note that the merging of two traces does not always yield a unique trace due to possible interleaving of actions outside the synchronization alphabet. Furthermore, merging two arbitrary traces t_1 and t_2 may not allow the combination of all actions from t_1 and t_2 into the resulting trace; it deals with the case that t_1 and t_2 cannot be merged completely (called "blocking") due to deadlock. However, since test derivation is based on particular maximal IO-df-chains in this paper, merging two traces should not lead to blocking in order to ensure that the maximal IO-df-chains are covered entirely. Therefore, we introduce an operator that determines whether two traces can be merged completely:

- two traces $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$ *match with respect to a* "*synchronization alphabet*" S, denoted by $\langle a_1, \ldots, a_n \rangle \cong_S \langle b_1, \ldots, b_n \rangle$, iff $\langle a_1, \ldots, a_n \rangle \lfloor S = \langle b_1, \ldots, b_n \rangle \lfloor S$ and for all a_i, $b_i \in S$, $(1 \leq i \leq n)$, the associated experiment offer instances satisfy the matching conditions according to the LOTOS dynamic semantics [16].

Trace Derivation System

As mentioned earlier, the operational semantics of LOTOS is defined in terms of axioms and inference rules [16]. These axioms and inference rules enable us to derive the observable and internal actions that a behaviour expression can perform. As such, they define a labelled transition relation $B \text{ –}a\rightarrow B'$, describing that the behaviour expression B is able to perform the (possibly unobservable) action a and then behave like B'. Based on this relation some other relations on labelled transition systems (LTSs) are defined [12]: [3]

$B \text{ –}a_1 \ldots a_n\rightarrow B'$ iff $\exists B_i\ (0 \leq i \leq n), B = B_0 \text{ –}a_1\rightarrow B_1 \text{ –}a_2\rightarrow B_2 \ldots \text{–}a_n\rightarrow B_n = B'$

$B = \langle a_1, \ldots, a_n \rangle \Rightarrow B'$ iff $\exists \mathrm{i}^{k_0} a_1 \mathrm{i}^{k_1} a_2 \mathrm{i}^{k_2} \ldots a_n \mathrm{i}^{k_n}\ (k_i \geq 0), B \text{ –}\mathrm{i}^{k_0} a_1 \mathrm{i}^{k_1} a_2 \mathrm{i}^{k_2} \ldots a_n \mathrm{i}^{k_n}\rightarrow B'$

The purpose of the *trace relation* "$\Rightarrow$" is to abstract from the unobservable actions that are derived by the trace derivation system. The guided inference rules, which are to be presented, make extensive use of this relation.

3.2.2 Guided Inference Rules

The possible traces that can be derived from a behaviour expression B may be limited by imposing an ASDP on B . It is precisely in this manner that ASDPs guide the derivation of traces from B. The imposition of an ASDP s on B is denoted as (s, B) and describes the derivation of B on some trace that reflects s. This is similar to the notation $(s, B)/A$ defined in [14], with the exception of s being an ASDP as opposed to an SDP and the omission of the set A of restricted actions. The set of restricted actions A is introduced to further limit the possible derivations of a behaviour expression to those complying with a certain SDP and not having any action in A. However, these additional restrictions on actions are currently of no interest for our test selection method and are therefore omitted in the notation.

We adapt the original inference rules to guided inference rules based on the semantics of (s, B). Accordingly, the guided inference rules differ from the usual inference rules in the following respects:

- they define the derivation of a behaviour expression not only on a single action, but also on traces;

[3] $\mathbf{i}^k$ denotes a sequence of k **i**-actions.

- they do not describe all possible derivations of a behaviour expression in general, but only those that comply with the ASDP imposed on the behaviour expression.

The guided inference rules are given in Table 1. In order to explain the general concept, consider the first guided inference rule for the choice operator (rule 7). This rule states that if an ASDP directs to the left of the choice operator in B_1 [] B_2 (indicated by [*ch*^*l*.*s*]), then the resulting trace must be a trace t of B_1 guided by the remainder ASDP s and the resulting behaviour must be B_1', i.e. the behaviour of B_1 after t. As another explanation, rule 4 (action prefix) states that if t is a trace of behaviour B guided by ASDP s, then $gd_1 \ldots d_n.t$ is a trace of $gd_1 \ldots d_n[P]$; B guided by ASDP [pf.s], provided that g is observable and that P evaluates to true. In other words, rule 4 accumulates the observable actions in a derived trace. This rule is complemented by rule 5, which skips the internal actions.

As a concrete example, consider the ASDP [hd, par([inst, pf, pf], [inst, pf, ch^r, pf])] corresponding to the maximal IO-df-chain in Fig. 4 (see subsection 3.1.2). Note that this is just one possible ASDP, namely the ASDP which considers zero occurrences of each possible path repetition. Let B be the behaviour expression given in Fig. 4, then the following relation must be satisfied to derive the test sequence t that covers the maximal IO-df-chain:

([hd, par([inst, pf, pf], [inst, pf, ch^r, pf])], B) $=t\Rightarrow$ B'

This relation matches rule 6, where the following relation must be satisfied ($t = t'\rceil\{\text{mid}\}$):

([par([inst, pf, pf], [inst, pf, ch^r, pf])], Input[in, mid] |[mid]| Output[mid, out]) $=t'\Rightarrow$ B'

This relation then matches rule 14, where the following relations must be satisfied ($t' = t_1|\{\text{mid}\}|t_2$):

1) ([inst, pf, pf], Input[in, mid]) $=t_1\Rightarrow$ B_1'
2) ([inst, pf, ch^r, pf], Output[mid, out] $=t_2\Rightarrow$ B_2'

Applying rule 15 to the first relation yields $t_1 = t_3$[in/a, mid/b] and $B_1' = B_3'$[in/a, mid/b], where ([pf, pf], a?x:Coin; b!x; Input[a, b]) $=t_3\Rightarrow$ B_3'. Twice applying rule 4 leads to $t_3 = \langle$a?x:Coin, b!$x\rangle$ and B_3' = Input[a, b]. This gives $t_1 = \langle$in?x:Coin, mid!$x\rangle$ and B_1' = Input[in, mid]. Similarly, the second relation gives $t_2 = \langle$mid?y:Coin, out!(2*y)$\rangle$ and B_2' = Output[mid, out]. Finally, we obtain $B' = B_1'$ |[mid]| B_2' and $t = (t_1 |\{\text{mid}\}| t_2)\rceil\{\text{mid}\}$, i.e.

t = ($\langle$in?x:Coin, mid!$x\rangle$ |{mid}| $\langle$mid?y:Coin, out!(2*y))$\rangle\rceil\{\text{mid}\}$
= $\langle$in?x:Coin, mid!x, out!(2*x)$\rangle\rceil\{\text{mid}\}$
= $\langle$in?x:Coin, out!(2*x)$\rangle$

Table 1 – Guided inference rules for LOTOS

base case	$([], B) = \langle\rangle \Rightarrow B$	(1)
unspecified case	$\dfrac{B = t \Rightarrow B'}{(_, B) = t \Rightarrow B'}$	(2)
successful termination	$([\text{ex}], \textbf{exit}(E_1, ..., E_n)) = \langle \delta_{<\text{eval}(E_1), ..., \text{eval}(E_n)>} \rangle \Rightarrow \textbf{stop}$	(3)
action prefix	$\dfrac{((s, B) = t \Rightarrow B',\ g \neq \textbf{i},\ P = \text{true})}{([\text{pf}.s], g d_1 ... d_n [P]; B) = \langle g d_1 ... d_n . t \rangle \Rightarrow B'}$	(4)
	$\dfrac{(s, B) = t \Rightarrow B'}{([\text{pf}.s], \textbf{i}; B) = t \Rightarrow B'}$	(5)
hiding	$\dfrac{(s, B) = t \Rightarrow B'}{([\text{hd}.s], \textbf{hide}\ D\ \textbf{in}\ B) = t \lceil \{D\} \Rightarrow \textbf{hide}\ D\ \textbf{in}\ B'}$	(6)
choice	$\dfrac{((s, B_1) = t \Rightarrow B_1',\ s \neq [])}{([\text{ch}\hat{}1.s], B_1\ []\ B_2) = t \Rightarrow B_1'}$	(7)
	$\dfrac{((s, B_2) = t \Rightarrow B_2',\ s \neq [])}{([\text{ch}\hat{}\text{r}.s], B_1\ []\ B_2) = t \Rightarrow B_2'}$	(8)
sequential composition	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ t_1\hat{} \neq \delta_{<EL>})}{([\text{en}(s_1, s_2)], B_1 >> \textbf{accept}\ IL\ \textbf{in}\ B_2) = t_1 \Rightarrow B_1' >> \textbf{accept}\ IL\ \textbf{in}\ B_2}$	(9)
	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ (s_2, B_2) = t_2 \Rightarrow B_2',\ t_1\hat{} = \delta_{<EL>})}{([\text{en}(s_1, s_2)], B_1 >> \textbf{accept}\ IL\ \textbf{in}\ B_2) = t_1 \lceil \{\delta\} \circ t_2 [RN] \Rightarrow B_2'[RN]}$ where $IL = x_1{:}s_1, ..., x_n{:}s_n$, $EL = \text{eval}(E_1), ..., \text{eval}(E_n)$, and $RN = E_1 / x_1, ..., E_m / x_m$	(10)
disabling	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ t_1\hat{} = \delta)}{([\text{dis}(s_1, s_2)], B_1\ [>\ B_2) = t_1 \Rightarrow B_1'}$	(11)
	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ t_1\hat{} \neq \delta)}{([\text{dis}(s_1, [])], B_1\ [>\ B_2) = t_1 \Rightarrow B_1'\ [>\ B_2}$	(12)
	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ (s_2, B_2) = t_2 \Rightarrow B_2',\ t_1\hat{} \neq \delta,\ s_2 \neq [])}{([\text{dis}(s_1, s_2)], B_1\ [>\ B_2) = t_1 \circ t_2 \Rightarrow B_2'}$	(13)
parallel composition	$\dfrac{((s_1, B_1) = t_1 \Rightarrow B_1',\ (s_2, B_2) = t_2 \Rightarrow B_2',\ t_1 \cong_{S \cup \{\delta\}} t_2)}{([\text{par}(s_1, s_2)], B_1\ \lvert[S]\rvert\ B_2) = t_1 \lvert S \rvert t_2 \Rightarrow B_1'\ \lvert[S]\rvert\ B_2'}$	(14)
process instantiation	$\dfrac{(\exists P[h_1, ..., h_n](x_1{:}s_1, ..., x_m{:}s_m) := B,\ (s, B) = t \Rightarrow B')}{(([\text{inst}.s], P[g_1, ..., g_n](E_1, ..., E_m)) = t[RN] \Rightarrow B'[RN])}$ where $RN = g_1 / h_1, ..., g_n / h_n, E_1 / x_1, ..., E_m / x_m$	(15)
guarded behaviour	$\dfrac{((s, B) = t \Rightarrow B',\ P = \text{true})}{([\text{gd}.s], [P] \rightarrow B) = t \Rightarrow B'}$	(16)
local definition	$\dfrac{(s, [E_1 / x_1, ..., E_n / x_n]B) = t \Rightarrow B'}{([\text{let}.s], \textbf{let}\ x_1{:}s_1 = E_1, ..., x_n{:}s_n = E_n\ \textbf{in}\ B) = t \Rightarrow B'}$	(17)
sum expression	$\dfrac{(s, [E_1 / x_1, ..., E_n / x_n]B) = t \Rightarrow B'}{([\text{sum}.s], \textbf{choice}\ x_1{:}s_1, ..., x_n{:}s_n\ []\ B) = t \Rightarrow B'}$ where E_i is a value expression of sort s_i $(1 \leq i \leq n)$	(18)

4 An Application

In this section, we apply the presented test selection method to the ABP specification given in Annex A which is adopted from the one given in [2]. Annex B shows the flowgraph representation of the specification (Fig. B.1 and Table B.1-B.2) and the classification of the variables employed in the specification (Table B.3-B.4). In addition, it gives the generated maximal IO-df-chains (Table B.5).

We work out in detail the generation of two test sequences, covering the maximal IO-df-chains $C1$ and $C2$ of the ABP specification, respectively. Test sequences for the other maximal IO-df-chains of the ABP specification are given in Table B.6 of Annex B. The APEs for $C1$ and $C2$ are given by

$$\mathrm{APE}_{C1} = (2,3,4,5,6,7,8,9,70,((37,38,39,40)^* \mid (37,38,41,42)^*)^*,37,38,41,72,55,56,57,58,59,(63,66,67,53)^*,63,64,65,(68,69,51)^*,68)$$

$$\mathrm{APE}_{C2} = (2,3,4,33,3,4,5,6,7,8,9,70,((37,38,39,40)^* \mid (37,38,41,42)^*)^*,37,38,41,72,56,57,58,59,(63,66,67,53)^*,63,64,65,(68,69,51)^*,68)$$

Table 2 summarizes the transformation of these APEs into a set of SDPs. For maximal IO-df-chain $C1$, transforming the possible SDPs into ASDPs and subsequently unifying the ASDPs yields

$$\mathrm{ASDP}_{C1} = [\mathrm{par}([\mathrm{inst},\mathrm{pf},\mathrm{en}([\mathrm{inst},\mathrm{hd},\mathrm{par}([\mathrm{pf},\mathrm{pf}],_)],_)],[\mathrm{par}(\mu(s_1),[\mathrm{inst}.un(\mu(s_2),un(\mu(s_3),\mu(s_4)))])])]$$

where

s_1 = [par^l,inst,((pf,ch^l,pf,inst)* | (pf,ch^r,pf,inst)*)*,pf,ch^r,pf]
s_2 = [(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^l,inst,pf,ch^l,gd,pf,ex]
s_3 = [(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^r,ch^l,gd,ex]
s_4 = [(en^r,pf,inst)*,en^r,pf]

and $un(s_1, s_2)$ denotes the unification of ASDPs s_1 and s_2.

The test sequence is now derived by satisfying the relation $(\mathrm{ASDP}_{C1}, B) =t_{C1}\Rightarrow B'$, where B is the behaviour of the ABP specification. This succeeds on the first attempt using s_1 = [par^l,inst,pf, ch^r,pf], s_2 = [en^l,inst,en^l,inst,pf,ch^l,gd,pf,ex], s_3 = [en^l,inst,en^r,ch^l,gd,ex], and s_4 = [en^r,pf], i.e. with the shortest possible ASDP_{C1}, and[4]

t_{C1} = ⟨*sendgate* ?datamsg : datamsg, *sendpdu* !makepdu(datamsg, @sseq), *receivepdu* !makepdu(datamsg, @sseq), *sendack* !makeack(@sseq), *receivegate* !datamsg⟩

[4] In the derivation of the executable tests, symbolic values are used for the initializations of the variables "seqsender" and "seqreceiver" rather than the specified groundterm "0 **of** Bit", because variable initializations through process instantiations are considered as inputs (see 2.2) which have a parameter variation field.

In this test no PDU loss occurs. Therefore, it yields the data flow which corresponds to the normal operation of the ABP, i.e. sending data by means of a PDU, receiving the PDU without any retransmission, and transferring the data to the user. Note that a symbolic value "@sseq" is used rather than the ground term "0" given in the specification. This is in accordance with the concept of input defined in Section 2.2, which states that variables initialized through process instantiations by values consisting of only ground terms are considered as inputs. These variables should thus be allowed to vary over their input domain.

Table 2 – Transforming two maximal IO-df-chains of the ABP into SDPs

APE *P*	**Split_path_ expression(*P*)**	**Rooted path expressions**	**SDPs**
P_{C1}	(2,3,4,5,6,7,8,9)	(1,2,3,4,5,6,7,8,9)	[par^l,inst,pf,en^l,inst,hd,par^l,pf,pf]
	(((37,38,39,40)* | (37,38,41,42)*)*,37,38,41)	(1,34,35,36,((37,38,39,40)* | (37,38,41,42)*)*,37,38,41)	[par^r,par^l,par^l,inst,((pf,ch^l,pf,inst)* | (pf,ch^r,pf,inst)*)*,pf,ch^r,pf]
	(55,56,57,58,59)	(1,34,50,51,(68,69,51)*,52,53,(63,66,67,53)*,54,55,56,57,58,59)	[par^r,par^r,inst,(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^l,inst,pf,ch^l,gd,pf,ex]
	((63,66,67,53)*,63,64,65)	(1,34,50,51,(68,69,51)*,52,53,(63,66,67,53)*,63,64,65)	[par^r,par^r,inst,(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^r,ch^l,gd,ex]
	((68,69,51)*,68)	(1,34,50,51,(68,69,51)*,68)	[par^r,par^r,inst,(en^r,pf,inst)*,en^r,pf]
P_{C2}	(2,3,4,33,3,4,5,6,7,8,9)	(1,2,3,4,33,3,4,5,6,7,8,9)	[par^l,inst,pf,en^r,inst,pf,en^l,inst,hd,par^l,pf,pf]
	(((37,38,39,40)* | (37,38,41,42)*)*,37,38,41)	(1,34,35,36,((37,38,39,40)* | (37,38,41,42)*)*,37,38,41)	[par^r,par^l,par^l,inst,((pf,ch^l,pf,inst)* | (pf,ch^r,pf,inst)*)*,pf,ch^r,pf]
	(55,56,57,58,59)	(1,34,50,51,(68,69,51)*,52,53,(63,66,67,53)*,54,55,56,57,58,59)	[par^r,par^r,inst,(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^l,inst,pf,ch^l,gd,pf,ex]
	((63,66,67,53)*,63,64,65)	(1,34,50,51,(68,69,51)*,52,53,(63,66,67,53)*,63,64,65)	[par^r,par^r,inst,(en^r,pf,inst)*,en^l,inst,(en^r,ch^r,gd,inst)*,en^r,ch^l,gd,ex]
	((68,69,51)*,68)	(1,34,50,51,(68,69,51)*,68)	[par^r,par^r,inst,(en^r,pf,inst)*,en^r,pf]

Similar to the above, we find the test sequence for $C2$ by solving $(\text{ASDP}_{C2}, B) =t_{C2}\Rightarrow B'$. This time, however, choosing the shortest possible ASDP_{C2} will make the guided inference rules fail in the attempt to derive a test sequence, due to the recursive process instantiation represented in the first SDP (i.e. [par^l,inst,pf,en^r,*inst*,pf,en^l,inst,hd,par^l,pf,pf]). In fact, the test derivation only succeeds if one uses s_1, s_2, s_3, and s_4 that take into account the effect of the recursive instantiation in this SDP. Therefore, a test sequence covering $C2$ can be generated by using

s_1 = [par^l,inst,pf,ch^r,pf,inst,pf,ch^r,pf]

s_2 = [en^r,pf,inst,en^l,inst,en^l,inst,pf,ch^l,gd,pf,ex]

s_3 = [en^r,pf,inst,en^l,inst,en^r,ch^l,gd,ex]

s_4 = [en^r,pf,inst,en^r,pf]

Transforming the given SDPs into ASDPs and further unifying the latter yields an $ASDP_{C2}$ that leads to the following test sequence:

t_{C2} = ⟨*sendgate* ?datamsg : datamsg, *sendpdu* !makepdu(datamsg, @sseq), *receivepdu* !makepdu(datamsg, @sseq), *sendack* !makeack(@sseq), *receivegate* !datamsg, *receiveack* !makeack(@sseq), *sendgate* ?datamsg : datamsg, *sendpdu* !makepdu(datamsg, Succ(@sseq)), *receivepdu* !makepdu(datamsg, Succ(@sseq)), *sendack* !makeack(Succ(@sseq)), *receivegate* !datamsg⟩

Again, no PDU loss occurs in this test. However, as oposed to t_{C1} data is sent successfully two consecutive times. Thus, t_{C2} identifies the data flow which corresponds to two successful PDU deliveries, thereby checking incrementation of the sequence numbers.

It should be observed that t_{C2} extends t_{C1}. As a consequence, t_{C1} could be disregarded. The number of tests would be optimized and all the maximal IO-df-chains of the ABP specification would still be covered.

5 Conclusions

We have presented a data flow oriented test selection method for LOTOS. The method basically consists of two phases. The first phase involves the application of static data flow analysis [10]. A given LOTOS specification is represented by a flowgraph which enables us to identify the functionality of the intended system in terms of the associations between each output and those inputs that influence the output. A set of maximal IO-df-chains is then derived to cover all these input-output relations. However, not all maximal IO-df-chains may correspond to sequences of actions which are feasible, due to the possible existence of rendez-vous. This obstacle is dealt with in the second phase. In this phase, the maximal IO-df-chains are transformed into guides which are used by a new set of inference rules for LOTOS to generate a set of feasible tests. This set of tests covers all the feasible maximal IO-df-chains of the specification.

The set of tests obtained by applying the proposed method provides the capability of determining whether an implementation under test (IUT) establishes the desired associations among values of input interaction parameters and output interaction parameters. Thus, the method complements control flow oriented test selection strategies for LOTOS. This raises an interesting topic for further research: a test selection method which considers both control flow

and data flow aspects of an IUT.

The second phase of our test selection method is based on the guided inference rules for LOTOS given in Table 1. Two of these rules (rule 4 and rule 16) imply the evaluation of guards and selection predicates. As we handle data values in a symbolic manner, it may happen that a predicate cannot be evaluated at the time of application of rule 4 or rule 16. In principle, these inference rules cannot be applied in that case. However, the idea is to keep the predicate as a condition for the resulting trace since the rendez-vous mechanism may provide values which could lead to the evaluation of the predicate at a later stage in the derivation of the test sequence. When there are still unevaluated predicates at the time the test sequence has been derived completely, the actual tester must provide appropriate values manually. If such values are impossible the derived test sequence is declared infeasible. It is clear that the above process requires a system for equation solving, e.g. a theorem prover.

In Section 4 we briefly addressed the issue of finding proper SDPs such that the guided inference rules succeed in deriving a particular test. This requires dealing with complications which are inherent to recursion and parallelism. In general, it is essential to determine precisely the number of iterations of loops in parallel components to handle concurrency. Adaptive strategies are needed for this purpose. In [5], an idea is presented to facilitate the determination of the required number of recursive instantiations for processes in arbitrary LOTOS specifications. It may direct further research needed on this topic.

Finally, we would like to stress that a test selection method may be considered as consisting of two distinct phases, the test selection phase and the test data selection phase. Although the test selection phase for LOTOS specifications has been discussed in detail with respect to the maximal IO-df-chains criterion, the selection of a particular collection of test data that will cause the execution of an IUT along the generated tests has not been addressed and requires further investigation.

Acknowledgements

This work is supported in part by the Telecommunications Research Institute of Ontario and in part by the Natural Sciences and Engineering Research Council of Canada. The authors wish to thank E. Brinksma, L. Logrippo, M. Haj-Hussein and J. Sincennes, for many valuable discussions and suggestions.

References

[1] F. Belina and D. Hogrefe, “The CCITT-specification and description language SDL”, *Comput. Networks and ISDN Syst.*, vol. 16, pp. 311-341, 1989.

[2] P. Berlinguette and D. Gueraichi, “The alternating bit protocol in LOTOS: "Textual" and "Graphical" representations”, Technical Report TR-88-25, Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada, 1988.

[3] T. Bolognesi and E. Brinksma, “Introduction to the ISO specification language LOTOS”, *Comput. Networks and ISDN Syst.*, vol. 14, no. 1, pp. 25-59, 1987.

[4] E. Brinksma, “A theory for the derivation of tests”, in *Protocol Specification*, *Testing*, *and Verification*, vol. 8, S. Aggarwal and K. Sabnani, Eds. The Netherlands, Amsterdam: North-Holland, 1988, pp. 63-74.

[5] E. Brinksma and H. Eertink, “Goal-driven LOTOS execution”, in *Protocol Specification*, *Testing*, *and Verification*, vol. 13, A. Danthine, G. Leduc and P. Wolper, Eds. The Netherlands, Amsterdam: North-Holland, 1993, pp. .

[6] S. Budkowski and P. Dembinski, “An introduction to ESTELLE: A specification language for distributed systems”, *Comput. Networks and ISDN Syst.*, vol. 14, no. 1, pp. 3-23, 1987.

[7] W. Chun and P. D. Amer, “Test case generation for protocols specified in Estelle”, in *Proc. FORTE '90*, Madrid, Spain, J. Quemada, J. Mañas, and E. Vazquez, Eds., pp. 197-210, 1990.

[8] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar, “Formal methods for generating protocol conformance test sequences”, in *Proc. IEEE*, vol. 78, no. 8, pp. 1317-1325, 1990.

[9] J. P. Favreau and R. J. Linn, “Automatic generation of test scenario skeletons from protocol specifications written in ESTELLE”, in *Protocol Specification, Testing, and Verification*, vol. 6, G. v. Bochmann and B. Sarikaya, Eds., The Netherlands, Amsterdam: North-Holland, 1986, pp. 191-202.

[10] L. D. Fosdick and L. J. Osterweil, “Data flow analysis in software reliability”, *ACM Comput. Surv.*, vol. 8, no. 3, pp.305-330, 1976.

[11] P. Frankl and E. Weyuker, “An applicable family of data flow testing criteria”, *IEEE Trans. Software Eng.,* vol. SE-14, no. 10, pp. 1483-1498, 1988.

[12] S. Gallouzi, L. Logrippo, and A. Obaid, “An expressive trace theory for LOTOS”, in *Protocol Specification*, *Testing*, *and Verification*, vol. 11, B. Johnsson, J. Parrow, and B. Pehrson, Eds. The Netherlands, Amsterdam: North-Holland, 1991, pp. 159-175.

[13] D. Guerachi and L. Logrippo, “Derivation of test cases for LAP-B from a LOTOS specification”, in *Proc. FORTE '89*, Vancouver, B.C., Canada, S. T. Vuong, Ed., pp. 489-508, 1989.

[14] M. Haj-Hussein, L. Logrippo, and J. Sincennes, “Goal oriented execution for LOTOS,” in *Proc. FORTE '92*, Lannion, France, M. Diaz and R. Groz, Eds., pp. 305-320, 1992.

[15] W. E. Howden, *Functional program testing and analysis*. McGraw Hill, New York, (1987).

[16] Open Systems Interconnection, “LOTOS - A formal description technique based on the temporal ordering of observational behaviour”, International Standard IS 8807, 1989.

[17] K. Knight, “Unification: A multidisciplinary survey”, *ACM Comput. Surv.*, vol. 21, no. 1, pp. 93-124, 1989.

[18] B. Korel, “The program dependence graph in static program testing”, *Info. Processing Letters*, vol. 24, pp. 103-108, 1987.

[19] J. W. Laski and B. Korel, “A data-flow oriented program testing strategy”, *IEEE Trans. Software Eng.,* vol. SE-9, no. 5, pp. 347-354, 1983.

[20] R.E. Miller and S. Paul, “On generating test sequences for combined control and data flow for conformance testing of communication protocols”, in *Protocol Specification*, *Testing*, *and Verification*, vol. 12, J. Linn and U. Uyar, Eds., The Netherlands, Amsterdam: North-Holland, 1992, pp. 1-15.

[21] S. C. Ntafos, “On required element testing”, *IEEE Trans. Software Eng.,* vol. SE-10, pp. 795-803, 1984.

[22] D. H. Pitt and D. Freestone, “The derivation of conformance tests from LOTOS specifications”, *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 1337-1343, 1990.

[23] K. K. Sabnani and A. T. Dahbura, “A protocol test generation procedure”, *Comput. Networks and ISDN Syst.*, vol. 15, pp. 285-297, 1988.

[24] H. van der Schoot, “Validation Activities for LOTOS based on Static Data Flow Analysis”, Memoranda Informatica 93-36, Department of Computer Science, University of Twente, Enschede, The Netherlands, 1993.

[25] H. van der Schoot and H. Ural, “Validation activities based on static data flow analysis of system specifications in LOTOS”, Technical Report TR-92-42, Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada, 1992. (submitted for publication)

[26] D. Sidhu and T-K. Leung, "Formal methods for protocol testing : A detailed study", *IEEE Trans. Software Eng.*, vol. SE-15, no. 4, pp. 413-426, 1989.
[27] D. Sidhu, "Protocol testing", in *Protocol Specification, Testing, and Verification*, vol. 10, L. Logrippo, R.L. Probert, and H. Ural, Eds., The Netherlands, Amsterdam: North-Holland, 1990, pp. 47-68.
[28] P. Tripathy and B. Sarikaya, "Test generation from LOTOS specifications", *IEEE Trans. Comp.*, vol. 40, no. 4, pp. 543-552, 1991.
[29] H. Ural and B. Yang, "A test sequence selection method for protocol testing", *IEEE Trans. Commun.*, vol. 39, no. 4, pp. 514-523, 1991.
[30] H. Ural, "Test sequence selection based on static data flow analysis", *Comput. Commun.*, vol. 10, no. 5, pp. 234-242, 1987.
[31] H. Ural and B. Yang, "Test path selection via IP/O-chains coverage criteria", Technical Report TR-91-32, Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada, 1991.
[32] C. D. Wezeman, "The CO-OP method for compositional derivation of conformance testers", in *Protocol Specification, Testing, and Verification*, vol. 9, E. Brinksma, G. Scollo, and C.A. Vissers, Eds. The Netherlands, Amsterdam: North-Holland, 1989, pp. 145-158.

Annex A

```
specification alternate [sendgate, sendpdu, receivepdu, sendack, receiveack, receivegate]: noexit
library
   BitString, Boolean, Bit
endlib
type data is BitString renamedby
    sortnames datamsg for BitString
endtype (* data *)
type mess is data
    sorts message
    opns
        sequence        : message        →   Bit          (*sequence number*)
        makepdu         : datamsg, Bit   →   message
        makeack         : Bit            →   message
        is-equal        : Bit, Bit       →   Bool
        is-not-equal    : Bit, Bit       →   Bool
        datamessage     : message        →   datamsg
    eqns
        forall m, n : Bit, msg : datamsg
        ofsort Bit
           sequence(makeack(n)) = n;
           sequence(makepdu(msg,n)) = n;
        ofsort datamsg
           datamessage(makepdu(msg,n)) = msg
        ofsort Bool
           is-equal(n,m) = n eq m;
           is-not-equal(n,m) = n ne m;
endtype (* mess *)
type time is
    sorts tim
    opns
        timestart   :        →   tim
        expired     :        →   tim
        reset       :        →   tim
endtype (* tim *)

behaviour
(sender [sendgate,sendpdu,receiveack] (0 of Bit)
  |[sendpdu,receiveack]|
( (medium [sendpdu,receivepdu] ||| medium [sendack,receiveack])
    |[sendack,receivepdu]|
    receiver [receivegate,receivepdu,sendack] (0 of Bit)
   )
)
where
    process medium [ ingate, outgate] : noexit :=
    ingate ?msg : message;
    (   (i; medium [ingate,outgate])                              (* loss *)
        []
        outgate !msg; medium [ ingate,outgate]          (* regular delivery *)
      )
    )
    endproc (* medium *)
    process sender [sendgate,sendpdu,receiveack] (seqsender : Bit) : noexit :=
    sendgate ?datamsg : datamsg;
    (safemesser [sendpdu,receiveack] (datamsg, seqsender)
      >>
      sender [sendgate,sendpdu,receiveack] (Succ(seqsender))
    )
```

```
    where
        process safemesser [sendpdu,receiveack] (datamsg : datamsg, seqsender : Bit) : exit :=
        hide tm in
        (   (tm !timestart;
             sendpdu !makepdu(datamsg, seqsender);
             ( (messer [receiveack] (seqsender)
                  >>
                  tm !reset; exit
               )
               [>
               ( tm !expired; exit
                  >>
                  safemesser [sendpdu,receiveack] (datamsg, seqsender)
               )
             )
            )
            |[tm]|
            timer [tm]
        )
        where
            process messer [receiveack] (seqsender : Bit) : exit :=
            receiveack ?ack : message;
            ([is-equal (sequence(ack), seqsender)] → exit
             []
             [is-not-equal (sequence(ack), seqsender)] → messer [receiveack] (seqsender)
            )
            endproc (* messer *)
            process timer [tm] : exit :=
            tm !timestart;
            (i;
             tm ! expired; exit
             [>
             tm !reset; exit )
            )
            endproc (* timer *)
        endproc (* safemesser *)
    endproc (* sender *)
    process receiver [receivegate,receivepdu,sendack] (seqreceiver : Bit) : noexit :=
    safeacker [receivepdu,sendack] (seqreceiver)
    >> accept datamsg : datamsg in
    (receivegate !datamsg;
     receiver [receivegate,receivepdu,sendack] (Succ (seqreceiver))
    )
    where
        process safeacker [receivepdu,sendack] (seqreceiver: Bit) : exit(datamsg) :=
        acker [receivepdu,sendack] (seqreceiver)
        >> accept pdu : message, ok : Bool in
        ([ok] → exit (datamessage (pdu))
         [] [not (ok)] → safeacker [receivepdu,sendack] (seqreceiver)
        )
        where
            process acker [receivepdu,sendack] (seqreceiver : Bit) : exit (message, Bool) :=
            receivepdu ?pdu : message;
            ([is-equal (sequence(pdu),seqreceiver)] →     sendack ! makeack(seqreceiver);
                                                          exit(pdu,true)
             []
             [is-not-equal (sequence(pdu), seqreceiver)] →     sendack !makeack(Succ(seqreceiver));
                                                               exit(pdu,false)
            )
            endproc (* acker *)
        endproc (* safeacker *)
    endproc (* receiver *)
endspec (* alternate *)
```

Annex B

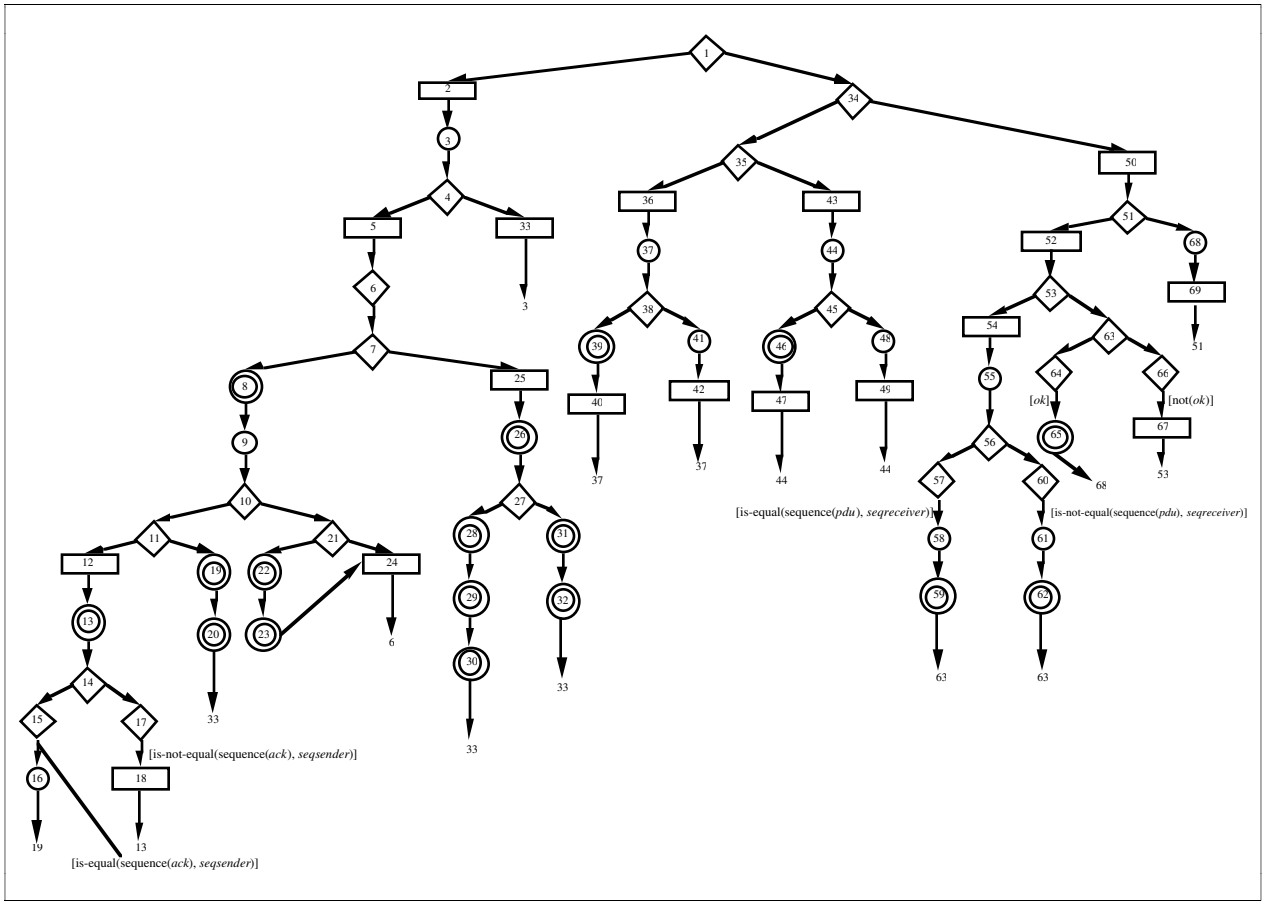

Fig. B.1 – The flowgraph of the ABP specification

Table B.1 – *s*-nodes and *s*-edges in the flowgraph of the ABP specification

***s*-nodes**	70	71	72	73	74
***s*-edges**	(9, 70) (70, 37)	(48, 71) (71, 13)	(41, 72) (72, 55)	(58, 73) (73, 44)	(61, 74) (74, 44)

Table B.2 – Labels of the nodes in the flowgraph of the ABP specification

Node	Label	Node	Label	Node	Label
1	|[*sendpdu*, *receiveack*]|	24	safemesser	47	medium
2	sender	25	timer	48	*receiveack*
3	*sendgate*	26	*tm*	49	medium
4	>>	27	[>	50	receiver
5	safemesser	28	**i**	51	>>
6	**hide**	29	*tm*	52	safeacker
7	|[*tm*]|	30	$\mathbf{i_{exit}}$	53	>>
8	*tm*	31	*tm*	54	acker
9	*sendpdu*	32	$\mathbf{i_{exit}}$	55	*receivepdu*
10	[>	33	sender	56	[]
11	>>	34	|[*sendack*, *receivepdu*]|	57	guard
12	messer	35	|||	58	*sendack*
13	*receiveack*	36	medium	59	$\mathbf{i_{exit}}$
14	[]	37	*sendpdu*	60	guard
15	guard	38	[]	61	*sendack*
16	$\mathbf{i_{exit}}$	39	**i**	62	$\mathbf{i_{exit}}$
17	guard	40	medium	63	[]
18	messer	41	*receivepdu*	64	guard
19	*tm*	42	medium	65	$\mathbf{i_{exit}}$
20	$\mathbf{i_{exit}}$	43	medium	66	guard
21	>>	44	*sendack*	67	safeacker
22	*tm*	45	[]	68	*receivegate*
23	$\mathbf{i_{exit}}$	46	**i**	69	receiver

Table B.3 – Renaming of variables in the ABP specification

variable	line number	renaming
seqsender	52	$seqsender_1$
seqsender	59	$seqsender_2$
seqsender	78	$seqsender_3$
seqreceiver	95	$seqreceiver_1$
seqreceiver	102	$seqreceiver_2$
seqreceiver	110	$seqreceiver_3$
datamsg	53	$datamsg_1$
datamsg	59	$datamsg_2$
msg	45	msg_1
msg	45	msg_2
pdu	111	pdu_1
pdu	104	pdu_2
ok	104	*ok*

Table B.4 – Defs and uses of variables for the ABP specification

Node/edge		**Node/edge**	
2	$d(seqsender_1)$	(60, 61)	$u(pdu_1)$ $u(seqreceiver_3)$
3	$d(datamsg_1)$	62	$u(pdu_1)$, $d(pdu_2)$ $d(ok)$
5	$u(datamsg_1)$, $d(datamsg_2)$ $u(seqsender_1)$, $d(seqsender_2)$	(64, 65)	$u(ok)$
12	$u(seqsender_2)$, $d(seqsender_3)$	65	$u(pdu_2)$, $d(datamsg_3)$
(15, 16)	$u(ack)$ $u(seqsender_3)$	(66, 67)	$u(ok)$
(17, 18)	$u(ack)$ $u(seqsender_3)$	68	$u(datamsg_3)$
33	$u(seqsender_1)$, $d(seqsender_1)$	69	$u(seqreceiver_1)$, $d(seqreceiver_1)$
50	$d(seqreceiver_1)$	70	$u(datamsg_2)$, $d(msg_1)$ $u(seqsender_2)$, $d(msg_1)$
52	$u(seqreceiver_1)$, $d(seqreceiver_2)$	71	$u(msg_2)$, $d(ack)$
54	$u(seqreceiver_2)$, $d(seqreceiver_3)$	72	$u(msg_1)$, $d(pdu_1)$
(57, 58)	$u(pdu_1)$ $u(seqreceiver_3)$	73	$u(seqreceiver_3)$, $d(msg_2)$
59	$u(pdu_1)$, $d(pdu_2)$ $d(ok)$	74	$u(seqreceiver_3)$, $d(msg_2)$

Table B.5 – Maximal IO-df-chains of the ABP specification

C1	A_1, B, E	C8	C, F
C2	A_2, B, E	C9	D_1, G
C3	A_3, B, E	C10	D_2, G
C4	A_1, B, F	C11	D_3, G
C5	A_2, B, F	C12	D_1, H
C6	A_3, B, F	C13	D_2, H
C7	C, E	C14	D_3, H

A_1 = $(dseqsender_1)_2$, $(useqsender_1)_5$ $(dseqsender_2)_5$

A_2 = $(dseqsender_1)_2$, $(useqsender_1)_{33}$ $(dseqsender_1)_{33}$, $(useqsender_1)_5$ $(dseqsender_2)_5$

A_3 = $(dseqsender_1)_2$, $(useqsender_1)_{33}$ $(dseqsender_1)_{33}$, $(useqsender_1)_{33}$ $(dseqsender_1)_{33}$, $(useqsender_1)_5$ $(dseqsender_2)_5$

B = $(useqsender_2)_{70}$ $(dmsg_1)_{70}$, $(umsg_1)_{72}$ $(dpdu_1)_{72}$

C = $(ddatamsg_1)_3$, $(udatamsg_1)_5$ $(ddatamsg_2)_5$, $(udatamsg_2)_{70}$ $(dmsg_1)_{70}$, $(umsg_1)_{72}$ $(dpdu_1)_{72}$

D_1 = $(dseqreceiver_1)_{50}$, $(useqreceiver_1)_{52}$ $(dseqreceiver_2)_{52}$, $(useqreceiver_2)_{54}$ $(dseqreceiver_3)_{54}$

D_2 = $(dseqreceiver_1)_{50}$, $(useqreceiver_1)_{69}$ $(dseqreceiver_1)_{69}$, $(useqreceiver_1)_{52}$ $(dseqreceiver_2)_{52}$, $(useqreceiver_2)_{54}$ $(dseqreceiver_3)_{54}$

D_3 = $(dseqreceiver_1)_{50}$, $(useqreceiver_1)_{69}$ $(dseqreceiver_1)_{69}$, $(useqreceiver_1)_{69}$ $(dseqreceiver_1)_{69}$, $(useqreceiver_1)_{52}$ $(dseqreceiver_2)_{52}$, $(useqreceiver_2)_{54}$ $(dseqreceiver_3)_{54}$

E = $(updu_1)_{59}$ $(dpdu_2)_{59}$, $(updu_2)_{65}$ $(ddatamsg_3)_{65}$, $(udatamsg_3)_{68}$

F = $(updu_1)_{62}$ $(dpdu_2)_{62}$, $(updu_2)_{65}$ $(ddatamsg_3)_{65}$, $(udatamsg_3)_{68}$

G = $(useqreceiver_3)_{73}$ $(dmsg_2)_{73}$, $(umsg_2)_{71}$

H = $(useqreceiver_3)_{74}$ $(dmsg_2)_{74}$, $(umsg_2)_{71}$

Table B.6 – Test Sequences for the ABP specification

t_{C1} `(* successful message delivery *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg⟩

t_{C2} `(* two consecutive successful message deliveries *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *receiveack*!makeack(@sseq), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receivegate*!datamsg⟩

t_{C3} `(* three consecutive successful message deliveries *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *receiveack*!makeack(@sseq), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receivegate*!datamsg, *receiveack*!makeack(Succ(@sseq)), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg⟩

t_{C9} `(* reception of an acknowledgement *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receiveack*!makeack(@sseq)⟩

t_{C10} `(* successful message delivery followed by the reception of an acknowledgement for the 2nd message *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *receiveack*!makeack(@sseq), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receiveack*!makeack(Succ(@sseq))⟩

t_{C11} `(* two successful message deliveries followed by the reception of an acknowledgement for the 3rd message *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *receiveack*!makeack(@sseq), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receivegate*!datamsg, *receiveack*!makeack(Succ(@sseq)), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receiveack*!makeack(@sseq)⟩

t_{C12} `(* reception of an out-of-sequence message (bits unequally initialized) *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receiveack*!makeack(@sseq)⟩

t_{C13} `(* correct message reception followed by the reception of an out-of-sequence message (e.g. due to a time-out) *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receiveack*!makeack(@sseq)⟩

t_{C14} `(* two correct message receptions followed by the reception of an out-of-sequence message (e.g. due to a time-out) *)`
⟨*sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,@sseq), *receivepdu*!makepdu(datamsg,@sseq), *sendack*!makeack(@sseq), *receivegate*!datamsg, *receiveack*!makeack(@sseq), *sendgate*?datamsg:datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receivegate*!datamsg, *sendpdu*!makepdu(datamsg,Succ(@sseq)), *receivepdu*!makepdu(datamsg,Succ(@sseq)), *sendack*!makeack(Succ(@sseq)), *receiveack*!makeack(Succ(@sseq))⟩