Data Flow Analysis of System Specifications in LOTOS

Hans van der Schoot and Hasan Ural

University of Ottawa
Department of Computer Science
Ottawa, Ontario, Canada K1N 9B4

Abstract

In LOTOS, a system is specified as a behaviour expression describing the externally observable behaviour of the system in terms of possible sequences of interactions between the system and its environment. The desired control flow and data flow that must be established by a possible implementation of the system are specified in the behaviour expression as implicit enumarations of allowed sequences of interaction identifiers and relationships among interaction parameters, respectively.

A model exposing the desired flow of data within the allowed control flow expressed in a system specification in LOTOS is presented. Based on the explicit information provided by the model, data flow anomaly detection and data flow oriented test selection are facilitated. A comprehensive example, i.e. an alternating bit protocol specification, is used to illustrate both these validation activities. An error in this specification is revealed by the analysis of a data flow anomaly detected within the specification. A set of test paths is derived from the specification by the application of an existing data flow oriented test selection criterion, called all-uses criterion.

Index Terms

Specification languages, modelling, software testing, data flow analysis, test generation

1. Introduction

LOTOS (Language Of Temporal Ordering Specification) is a standard formal description technique developed within ISO for specifying, among others, distributed concurrent information processing systems [2, 13, 15]. This language adopts most of its concepts from Milner's CCS [17] and Hoare's CSP [11]. It is based on process algebraic methods and uses the model of labelled transition systems as underlying semantics.

A system is specified in LOTOS as a behaviour expression describing the externally observable behaviour of the system in terms of possible sequences of interactions that the system exchanges with its environment. Each interaction can be viewed as an interaction identifier followed by a list of interaction parameters. Information is transferred to and from the system by means of these parameters. The implicit enumeration of possible interaction sequences of a system in a behaviour expression is in fact a description of the desired control flow and data flow that must be established by any implementation of the system. The control flow is described basically by the possible sequences of interaction identifiers, whereas the data flow is described by the relationships among the interaction parameters associated with the different interaction identifiers. In this paper it is demonstrated that the explicit identification of the desired flow of data within the allowed control flow expressed in a system specification in LOTOS supports static data flow analysis [6] based validation activities, namely, data flow anomaly detection and data flow oriented test selection.

A data flow anomaly (i.e. a variable which is defined but is not subsequently referenced) in a given specification may be considered as a symptom of possible errors (e.g. design omissions). The detection of data flow anomalies in a system specification should be an integral part of validation activities performed at this early stage of the development process, since corrective actions for specification errors that are not detected until implementation and (conformance) testing can be very costly.

Data flow oriented test selection has been promoted for testing the data flow aspects of a system implementation [7, 14, 18, 20] for conformance to its specification [23, 24]. The tests constructed from a system specification by data flow oriented test selection methods are considered complementary to those that are obtained by control flow oriented test selection methods, which concentrate on testing the conformance of control flow aspects of a system implementation. Although various control flow oriented test selection methods have been proposed for LOTOS [3, 4, 5, 8, 25], the applicability of (static) data flow analysis based validation techniques to system specifications in LOTOS has not been investigated.

Both data flow anomaly detection and data flow oriented test selection are based on the identification of the flow of data, that is the associations between definitions (i.e. defining occurrences [13]) of variables and uses (i.e. applied occurrences [13]) of variables, within the allowed control flow expressed in a system specification. In order to establish the associations between definitions and uses of variables in a system specification in LOTOS, occurrences of variables must be classified as definitions or uses and the relative position of these definitions and uses must be determined. The information required for both data flow anomaly detection and data flow oriented test selection is then made explicit by the proposed model, i.e. the flowgraph representation of a system specification in LOTOS.

The work presented in this paper has been stimulated to a large extent by traditional data flow analysis based software validation methods [6, 10, 12, 19, 26]. These methods were originally intended for programs written in procedural programming languages. Their applicability to concurrent programming languages which embody synchronization and communication constructs presents considerable challenges [21, 22] in terms of constraints placed on interprocess communication during the construction of a flowgraph representation. Like concurrent programming languages, LOTOS incorporates mechanisms for interprocess communication and synchronization. Furthermore, LOTOS includes other powerful operators, e.g. for disabling a behaviour expression, for enabling a behaviour expression, and for hiding observable actions in a behaviour expression. Such mechanisms and operators, which are typically absent in procedural programming languages, are handled by the rules proposed in this paper for the construction of a flowgraph representation of a system specification in LOTOS.

The rest of the paper is organized as follows. Section 2 discusses the data flow analysis of LOTOS specifications, defines the rules for constructing a flowgraph from a LOTOS specification, and presents the classification of the variable occurrences in a flowgraph as definitions and uses. As an example, section 3 gives the flowgraph representation of an alternating bit protocol specification in LOTOS. Section 4 discusses the application of the static data flow analysis based validation activities: data flow anomaly detection and data flow oriented test selection. Finally, section 5 contains the concluding remarks.

2. Data Flow Analysis of LOTOS Specifications

LOTOS specifications which are considered in this paper do not contain:

- recursive process instantiations with non-identical gate relabelling;
- recursive process instantiations which are not tail recursive[1];
- generalized parallel constructs (par expressions);
- generalized choice constructs (sum expressions) over gates.

These assumptions do not necessarily restrict in practice the applicability of data flow analysis to LOTOS. First of all, in order to obtain a recursive process instantiation with identical gate relabelling one can fall back on dynamic expansion; 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 end 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. This kind of recursion can always be avoided in the attempted definition of a process behaviour. Finally, the two generalized constructs mentioned above are only short-hands for other LOTOS expressions which are handled by the mapping of a behaviour expression to a flowgraph, as defined in the next section.

2.1 Mapping a Behaviour Expression to a Flowgraph

The flowgraph of a LOTOS specification is defined as the flowgraph of the behaviour part of the specification. Let "**specification** *S*[<gate list>] (<parameter list>) : <functionality> <data types> **behaviour** *B* **endspec**" be a definition of a LOTOS specification, where *S* is the specification identifier. The flowgraph of *S* is the flowgraph of *B* (henceforth denoted by $G(B)$). The transformation of *B* into $G(B)$ is defined as a mapping from *B* to a directed graph (N, E), where

$N = \{n \mid n$ is an o-node, i-node, p-node, c-node, or s-node$\}$, and

$E = \{e \mid e$ is an r-edge, s-edge, or e-edge$\}$.

Informally, the transformation of a behaviour expression *B* into the flowgraph $G(B)$ is such that:

- an *o*-node represents an *observable* action in *B*;
- an *i*-node represents an *internal* action in *B*;

1 In LOTOS a recursive process instantiation is not tail recursive if it occurs just before the enable or disable operator.

- a p-node represents a *process instantiation* in B;
- a c-node represents a nullary, unary (except for the action prefix), or binary *constructor* in B;
- an s-node represents a *synchronization* with value passing in B;
- an r-edge represents static control flow in B;
- an e-edge represents dynamic control flow caused by *enabling* in B;
- an s-edge represents dynamic control flow caused by a *synchronization* with value passing in B;

All the nodes in N except s-nodes are labelled: an o-node or i-node with the corresponding action denotation, a p-node with the corresponding process identifier, and a c-node with the corresponding constructor denotation. The flow of control between nodes is represented by edges. The distinction in E between edges representing static control flow (i.e. the r(egular)-edges) and edges representing dynamic control flow (i.e. the e(nable)-edges and the 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 [13], whereas the e-edges and s-edges reflect dynamic aspects of B [13]. The edges in E may have predicates associated with them. This is due to the fact that predicates affect both control flow and data flow and should therefore be incorporated in the flowgraph.

In this study, the definition of the flowgraph mapping is by recursion over LOTOS behaviour expressions, i.e. the flowgraph of a compound behaviour is defined in terms of the flowgraphs of its components. Therefore, it is understood that the former is not defined if any of the latter is not defined. To facilitate the mapping of a behaviour expression into a flowgraph, a single node in each flowgraph is identified as the *entry node*. The concept of entry node makes it possible to define the mapping in terms of separate rules for different syntactic LOTOS constructs. Each rule thus describes how the flowgraph of the LOTOS construct under consideration is built and defines a unique entry node for it. The formal definition of the flowgraph mapping G: $B \rightarrow < N, E >$ is given below, and is attended by the following notational conventions:

- L stands for the universe of (labels of) *observable* actions;
- B (as well as B_1, B_2, or B_p) stands for a LOTOS behaviour expression;
- **i** stands for the *internal* action;
- δ stands for the *successful termination* action;
- $\alpha(B) \subseteq L$ stands for the set of observable actions in B, i.e. the *alphabet* of B;
- d_i (as well as e_i) stands for an *experiment offer*, i.e. $d_i = ?x_i{:}s_i$ or $d_i = !E_i$, where $x_i{:}s_i$ denotes a variable x_i of sort s_i, and E_i denotes a value expression.

Inaction

The expression **stop** represents the completely inactive behaviour.

R1 G(**stop**) consists of a c-node with label *stop*. This node is also the entry node of G(**stop**).

Successful termination

The behaviour expression **exit**$(E_1,...,E_k)$ can perform the special, succesful termination action δ, and may thereby offer values $E_1,...,E_k$.

R2 G(**exit**$(E_1,...,E_k)$), where $k \geq 0$, consists of a c-node with label *exit*. This node is also the entry node of G(**exit**$(E_1,...,E_k)$).

Action prefix

The behaviour expression $ad_1...d_k[P];B$ can, subject to condition P (if any), engage in action a with associated input and/or output parameters (the possibly empty experiment offer list $d_1...d_k$), and then proceed as B.

R3 $G(ad_1...d_k[P];B)$, where $k \geq 0$ and $a \in L \cup \{\mathbf{i}\}$, is constructed by creating an o-node (when $a \in L$) or an i-node (when $a = \mathbf{i}$) with label a and with an outgoing r-edge to the entry node of $G(B)$; the predicate P (if any) is associated with this r-edge. The o-node or i-node with label a is the entry node of $G(ad_1...d_k[P];B)$.

Choice

The behaviour expression B_1 [] B_2 either behaves like B_1 or like B_2. Consequently, the flowgraph of B_1 [] B_2 has two subgraphs, namely $G(B_1)$ and $G(B_2)$.

R4 $G(B_1$ [] $B_2)$ is constructed by creating a c-node with label [] and with an outgoing r-edge to the entry node of $G(B_1)$ and an outgoing r-edge to the entry node of $G(B_2)$. The c-node with label [] is the entry node of $G(B_1$ [] $B_2)$.

Disabling

The behaviour expression B_1 [> B_2 describes a composition that allows B_2 to disable B_1 at any time, as long as B_1 has not terminated successfully. The variable scoping of B_1 [> B_2, as defined by the LOTOS static semantics [13], makes it impossible for a variable defined in B_1 to be referenced in B_2, and vice versa. Hence, it suffices to treat the disable operator like the choice operator during the flowgraph construction.

R5 $G(B_1 \,[\!> B_2)$ is constructed by creating a *c*-node with label $[\!>$ and with an outgoing *r*-edge to the entry node of $G(B_1)$ and an outgoing *r*-edge to the entry node of $G(B_2)$. The *c*-node with label $[\!>$ is the entry node of $G(B_1 \,[\!> B_2)$.

Enabling

The behaviour expression B_1 **>> accept** $x_1{:}s_1,\ldots,x_k{:}s_k$ **in** $B_2(x_1,\ldots,x_k)$ describes a sequential composition; it expresses the enabling of B_2 by B_1 if B_1 terminates successfully. When the list of variable declarations in the *accept* construct is not empty (that is when the accept construct is present) values are passed from B_1 to B_2.

R6 $G(B_1$ >> **accept** $x_1{:}s_1,\ldots,x_k{:}s_k$ **in** $B_2(x_1,\ldots,x_k))$, where $k \geq 0$, is constructed as follows:

- create a *c*-node with label >> and with an outgoing *r*-edge to the entry node of $G(B_1)$ and an outgoing *r*-edge to the entry node of $G(B_2)$. This *c*-node is the entry node of $G(B_1$ >> **accept** $x_1{:}s_1,\ldots,x_k{:}s_k$ **in** $B_2(x_1,\ldots,x_k))$,
- add an *e*-edge from all *c*-nodes in $G(B_1)$ with label *exit* to the entry node of $G(B_2)$, and replace these *c*-nodes by *i*-nodes with label i_{exit}.

Hiding

Hiding can transform some observable actions in a behaviour expression into internal actions, thereby making them unavailable for further synchronization.

R7 $G($**hide** D **in** $B)$, where $D \subseteq L$, is constructed by creating a *c*-node with label *hide* and with an outgoing *r*-edge to the entry node of $G(B)$, and by replacing every *o*-node in $G(B)$ with label $l \in D$ by an *i*-node with label l. The *c*-node with label *hide* is the entry node of $G($**hide** D **in** $B)$.

Process instantiation and relabelling

A process instantiation $P[a_1,\ldots,a_l](E_1,\ldots,E_k)$ refers to a process definition that must exist somewhere in the specification. A process definition packages behaviour just as a procedure packages statements in a conventional programming language. It may define fixed behaviour, or it may define a variety of behaviours according to parameters. Also, a process definition may instantiate itself in which case the instantiation is said to be *recursive*.

Let $G(B[a_1/a'_1,...,a_l/a'_l])$ denote the flowgraph $G(B)$ in which every o-node with label a'_j $(1 \leq j \leq l)$ has been assigned a new label a_j. Accordingly, $G(B)$ and $G(B[a_1/a'_1,...,a_l/a'_l])$ are two different graphs if there is at least one j such that $a_j \neq a'_j$.

R8 Let "**process** $P[a'_1,...,a'_l](x_1,...,x_k)$:= B_p **endproc**" be a process definition, where $k \geq 0$ and $l \geq 0$, then $G(P[a_1,...,a_l](E_1,...,E_k))$ is constructed as follows:

- if $P[a_1,...,a_l](E_1,...,E_k)$ is *not* a recursive instantiation, then create a p-node with label P and with an outgoing r-edge to the entry node of $G(B_p[a_1/a'_1,...,a_l/a'_l])$. The p-node with label P is the entry node for $G(P[a_1,...,a_l](E_1,...,E_k))$;
- if $P[a_1,...,a_l](E_1,...,E_k)$ is a recursive instantiation, then
  - if $\forall j$, $a_j = a'_j$, then create a p-node with label P and with an outgoing r-edge to the entry node of $G(B_p)$[2];
  - if $\exists j$ such that $a_j \neq a'_j$, then create a p-node with label P and with an outgoing r-edge to the entry node of $G(B_p[a_1/a'_1,...,a_l/a'_l])$.

The detection of recursion in this rule is only achieved for process instantiations which are free from non-identical gate relabelling. Recursion with non-identical gate relabelling cannot be detected statically. As an example consider the following process definition:

```
process P[a,b] := a;P[b,a] endproc
```

When process `P[a,b]` instantiates itself for the second time by means of '`P[b,a]`', this is in fact an instantiation '`P[a,b]`' due to dynamic relabelling. However, this cannot statically be detected, because the relabelling is necessarily done after completion of the flowgraph construction. G(`a;P[b,a][b/a, a/b]`) is constructed over and over again, corresponding to an infinite number of applications of rule R8. As a result, the flowgraph of the behaviour defined by the process causing the recursion is undefined, and thus the flowgraph of the instantiation itself is undefined.

---

2 $G(B_p)$ is already under construction, as the recursive instantiation $P[a_1,...,a_l](x_1,...,x_k)$ appears in B_p itself. This recursion is thus represented as such in the flowgraph.

Parallel composition

The behaviour expression B_1 |[A]| B_2 describes a composition in which B_1 and B_2 must synchronize on the actions in A. It can perform any action not in A that either B_1 or B_2 is ready to perform, or any action in $A \cup \{\delta\}$ that both B_1 and B_2 are ready to perform. B_1 and B_2 composed in parallel are said to affect each other in terms of the possible flow of data if the effect of synchronization is *value passing*. Only in this case data is exchanged between B_1 and B_2. On the other hand, if the effect of synchronization is either value matching or value generation then there is no explicit exchange of data, but there is only value agreement (see Table 1).

Table 1. Conditions for and effect of synchronization in LOTOS.

experiment offer d_i	**experiment offer e_i**	**condition**	**effect**
$!t_1$	$!t_2$	value(t_1) = value(t_2)	value matching
$!t$	$?x{:}s$	value(t) $\in s$	value passing
$?x_1{:}s_1$	$?x_2{:}s_2$	$s_1 = s_2$	value generation

R9 $G(B_1$ |[A]| $B_2)$, where $A \subseteq \alpha(B_1)\cup\alpha(B_2)$, is constructed as follows:

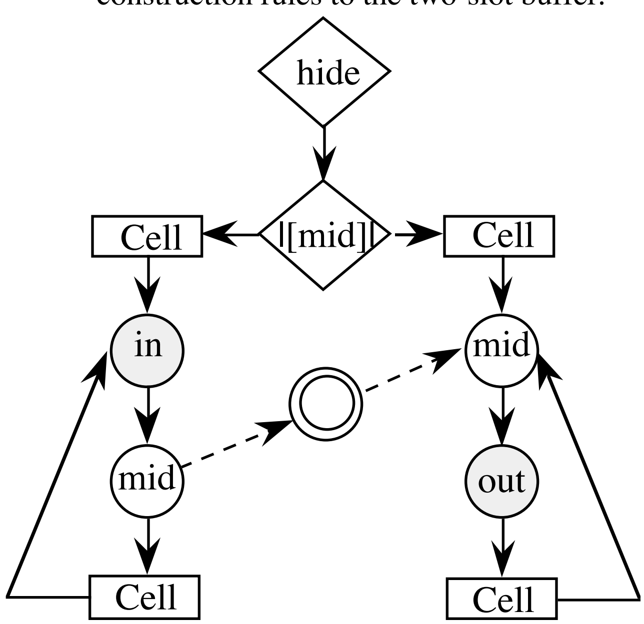
- create a c-node with label |||, |[A]|, or ||, when $A = \emptyset$, $\alpha(B_1)\cup\alpha(B_2) \supset A$, or $A = \alpha(B_1)\cup\alpha(B_2)$, respectively, and with an outgoing r-edge to the entry node of $G(B_1)$ and an outgoing r-edge to the entry node of $G(B_2)$. This c-node is the entry node for $G(B_1$ |[A]| $B_2)$,
- let n_1 be an o-node in $G(B_1)$ representing an (observable) action $ad_1...d_k$ and let n_2 be an o-node in $G(B_2)$ representing an action $ae_1...e_k$ (see rule R2, action prefix), such that k>0, $a \in A$, and d_i and e_i ($1 \le i \le k$) satisfy the synchronization conditions depicted in Table 1. For every pair of o-nodes (n_1, n_2), whose corresponding actions can potentially interact according to the semantics of the |[A]|-operator, do the following ($1 \le j \le k$):
  - i) if $\exists i\ d_i = !E_i$ and $e_i = ?x_i{:}s_i$ then create an s-node with an incoming s-edge from n_1 and an outgoing s-edge to n_2;
  - ii) if $\exists i\ d_i = ?x_i{:}s_i$ and $e_i = !E_i$ then create an s-node with an incoming s-edge from n_2 and an outgoing s-edge to n_1.

In the above rule case i) covers the situation where there is value passing from B_1 to B_2, and case ii) handles value passing in the opposite direction. Note that rule R9 excludes interactions via the succesful termination action δ since no value passing is involved.

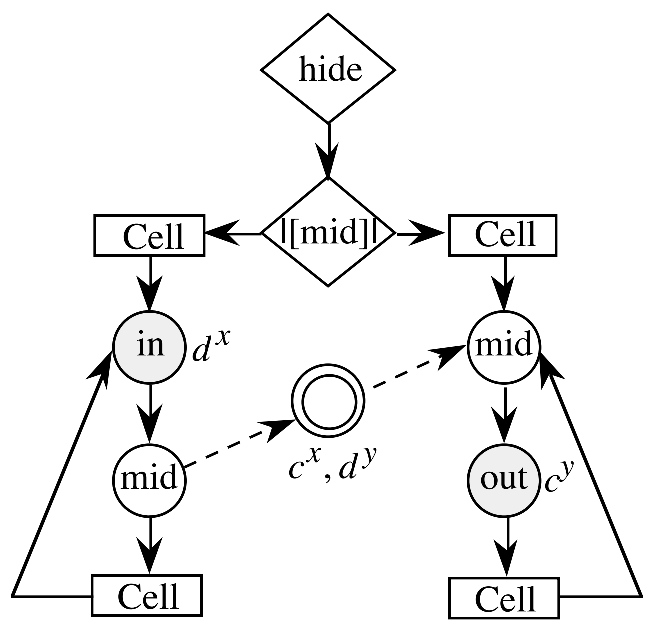
In the proposed analysis it can only be concluded that an interaction *may* cause value passing. The absence of the actual values of the interaction parameters makes it generally impossible to determine statically whether synchronization will really occur. Therefore, it is important to note that due to the static nature of the analysis, the *s*-nodes in a flowgraph represent *potential* interactions, which are derived according to R9 from the potential interactions implied in a behaviour expression. As an example, consider the following LOTOS description of a two-slot buffer:

```
hide mid in
     Cell[in,mid] |[mid]| Cell[mid,out]
where
     process Cell[a,b]:=
          a?x:nat; b!x; Cell[a,b]
     endproc
```

The corresponding flowgraph is depicted in Fig. 1.a. A diamond, rectangle, shaded circle, circle, double circle, normal edge and dotted edge represents a *c*-node, *p*-node, *o*-node, *i*-node, *s*-node, *r*-edge and *s*-edge, respectively. It should be observed that data may "flow" from the left `Cell` process to the right `Cell` process via interactions on the internal gate `mid`. The representation of this value passing is achieved in the flowgraph by the connection between the two *i*-nodes with label 'mid' (representing the respective actions in the two `Cell` processes) via an *s*-node and (directed) *s*-edges. In fact, rule R9 first joins the two actions `mid` as *o*-nodes, after which rule R7 replaces the *o*-nodes by *i*-nodes. The reader can easily verify the complete flowgraph in Fig. 1.a by applying the flowgraph construction rules to the two-slot buffer.



a



b

Fig. 1.   A Two-slot Buffer.

Guarded behaviour

A guard associated with a behaviour imposes a condition on the progress of that behaviour.

R10 $G([P]\rightarrow B)$ is constructed by creating a c-node with label *guard* and with an outgoing r-edge to the entry node of $G(B)$; P is associated with this r-edge. The c-node is the entry node for $G([P]\rightarrow B)$.

Local definition

A "let" construct allows the replacement of complex value expressions in a behaviour expression by variables.

R11 $G(\textbf{let}\ x_1 = E_1,\ldots,x_k = E_k\ \textbf{in}\ B(x_1,\ldots,x_k))$ $(k>0)$ is constructed by creating a c-node with label *let* and with an outgoing r-edge to the entry node of $G(B)$. The c-node is the entry node for $G(\textbf{let}\ x_1 = E_1,\ldots,x_k = E_k\ \textbf{in}\ B(x_1,\ldots,x_k))$.

Generalized choice

A "choice" construct (also: "summation") can be used to range over a possibly infinite set of values.

R12 $G(\textbf{choice}\ x_1{:}s_1,\ldots,x_k{:}s_k\ []\ B(x_1,\ldots,x_k))$ $(k>0)$ is constructed by creating a c-node with label *sum* and with an outgoing r-edge to the entry node of $G(B)$. The c-node is the entry node for $G(\textbf{choice}\ x_1{:}s_1,\ldots,x_k{:}s_k\ []\ B(x_1,\ldots,x_k))$.

2.2 Classification of Variables

Before variable occurrences are classified as defs and uses, the variables must be examined with respect to the scope of their declaration. A variable in LOTOS has local significance, i.e. the variable is unique only within a particular part of the specification which is defined by the scope of the declaration of the variable. Therefore, the same identifier can be used for several variables which serve different purposes. Renaming of such variables in the specification is necessary to avoid global conflicts that might arise in the flowgraph. After the variable renaming, the variable occurrences in the specification, occurring either as part of a syntactic element associated with a node in the flowgraph, or in a guard or selection predicate associated with an edge in the flowgraph, can be classified as definitions or uses. A *definition* (or *def*) of a variable x at node n (denoted by $(d_x)_n$) is an occurrence of x by which it takes a value, and a *use* of a variable x at node n (denoted by $(u_x)_n$) or on edge (n, m) (denoted by $(u_x)_{(n,m)}$) is an occurrence of x by which its value is referenced. In LOTOS, a def or a use of a variable is then interpreted as a variable occurrence in an 'identifier-declaration' or a 'value-expression', respectively [13]. As in [20], a use of a variable is further classified as a *computational use* (c-use, denoted by $(c_x)_n$) or a *predicate use* (p-use, denoted by $(p_x)_{(n,m)}$). A use of a variable in LOTOS is a p-use if the variable occurs in a guard or selection predicate, otherwise it is a c-use.

The following convention is used to identify defs, c-uses, and p-uses in the flowgraph $G(B)$ of behaviour expression B:

a. An o-node or i-node with label a, representing an action $ad_1...d_k$ which is not involved in any value passing[3], contains for $i = 1,...,k$
  - a def of x_i if $d_i = ?x_i{:}s_i$
  - a c-use of each variable in E_i if $d_i = !E_i$

b. A c-node with label *exit*, representing an exit construct "**exit**($E_1,...,E_k$)" without any subsequent enabling (see rule R6) contains a c-use of each variable in $E_1,...,E_k$,

c. An i-node with label i_{exit}, representing an exit construct "**exit**($E_1,...,E_k$)" followed by an accept construct "**accept** $x_1{:}s_1,...,x_k{:}s_k$ **in**" (see rule R6) contains a c-use of each variable in E_i followed by a def of x_i ($1 \leq i \leq k$),

d. A p-node with label P, representing a *non-recursive* instantiation $P[a_1,...,a_l](E_1,...,E_k)$ of a process definition "**process** $P[a'_1,...,a'_l](x_1,...,x_k)$:= B_p **endproc**"contains a c-use of each variable in E_i followed by a def of x_i ($1 \leq i \leq k$),

e. A p-node with label P, representing a *recursive* instantiation $P[a_1,...,a_l](E_1,...,E_k)$ of a process definition "**process** $P[a'_1,...,a'_l](x_1,...,x_k)$:= B_p **endproc**" contains a c-use of each variable in E_i followed by a def of x_i ($1 \leq i \leq k$) if $E_i \neq x_i$,

f. A c-node with label *let*, representing a local variable declaration construct "**let** $x_1 = E_1,...,x_k = E_k$ **in**" contains a c-use of each variable in E_i followed by a def of x_i ($1 \leq i \leq k$),

g. A c-node with label *sum*, representing a generalized choice construct "**choice** $x_1{:}s_1,...,x_k{:}s_k$ **[]**" contains defs of $x_1,...,x_k$,

h. An r-edge associated with a guard or selection predicate [$P(E_1,...,E_k)$] contains a p-use of each variable in $E_1,...,E_k$.

---

3 An o-node or i-node is involved in value passing when it has an incoming or outgoing s-edge.

In case of a *recursive* process instantiation it follows from e) that if $E_i = x_i$, the p-node in question does not contain a c-use of x_i before a def of x_i. The rationale here is that since the value of x_i has not changed, no new associations between a def of x_i and uses of x_i should be introduced.

s-nodes, created by rule R9 for parallel composition, are used to represent the potential interactions expressed in a given behaviour expression which may cause value passing. These nodes contain the definitions and uses of variables which can be identified in those experiment offers of actions not covered by a), that actually cause the value passing. Definitions and uses of the variables in the remaining experiment offers are still associated with their original nodes.

i. An s-node representing the interaction between two actions $ad_1...d_k$ and $ae_1...e_k$ contains a c-use of each variable in E_i followed by a def of x_i if $d_i = !E_i$ and $e_i = ?x_i{:}s_i$, or $d_i = ?x_i{:}s_i$ and $e_i = !E_i$ $(1 \le i \le k)$

j. An o-node (or i-node) representing an action $ad_1...d_k$, which is involved in value passing together with an o-node (or i-node) representing an action $ae_1...e_k$, contains for $i = 1,...,k$
   - a def of x_i if $d_i = ?x_i{:}s_i$ and $e_i = ?y_i{:}s_i$
   - a c-use of each variable in E_i if $d_i = !E_i$ and $e_i = !F_i$

It should be noted that j) complements a) for o-nodes and i-nodes. It also complements i) because i) covers value passing whereas j) covers the other two interaction types, i.e. value generation and value matching.

As an example, consider once more the LOTOS description of a two-slot buffer given in Section 2.1. Although at first it may look like there is only one variable involved, two different variables should be distinguished due to the two instantiations of the `Cell` process. One of the variables is renamed to avoid scope conflicts. The classification of the variable occurrences in the LOTOS description of the two-slot buffer is given in Fig. 1.b.

3. A Complete Example

In the appendix, a LOTOS specification of the alternating bit protocol (ABP) is given [1]. This protocol supports a reliable, uni-directional data transfer service between two users, the AB source and the AB sink. It uses an unreliable medium to send and receive AB PDUs and acknowledgements. In order to ensure that data is delivered in the correct order, the AB PDUs and acknowledgements carry a sequence number. The sequence numbers are calculated modulo 2, i.e. they alternate between 0 and 1.

The ABP specification is used as an example to illustrate the mapping from a LOTOS specification to a flowgraph and the classification of variables in a flowgraph.

3.1 The Flowgraph of the ABP

The flowgraph obtained by applying the mapping in section 2.1 to the ABP specification in the appendix is given in Fig. 2. For ease of presentation, the *s*-nodes and *s*-edges of this graph are not shown in Fig. 2 but listed in Table 2.

Table 2. *s*-nodes and *s*-edges in the flowgraph of the example ABP (see Fig.2).

***s*-nodes**	67	68	69	70	71
***s*-edges**	(10,67) (67,34)	(45,68) (68,14)	(38,69) (69,52)	(55,70) (70,41)	(58,71) (71,41)

3.2 Definitions and Uses of Variables in the ABP

Each variable occurrence in the ABP specification must be classified as a def, c-use, or p-use, and allocated to the nodes and edges in the flowgraph of the specification. In order to avoid global conflicts, variables with the same name but with a different scope are first renamed uniquely. The result of this renaming is shown in Table 3 where the variable msg in line number 46 is renamed twice, because of the two instantiations of process medium in line number 39. The classification of each variable occurrence in the flowgraph of the ABP specification as a def, c-use, or p-use is given in Table 4. Note that the *p*-nodes 21 and 64, representing recursive instantiations of processes "safemesser" and "safeacker" respectively (line numbers 71 and 104), do not contain any uses and defs since the values of the variables in these instantiations have not been changed (see case e), Section 2.2).

Table 3. Renaming of variables in the ABP specification.

variable	linenumber	renaming
seqsender	53	$seqsender_1$
seqsender	60	$seqsender_2$
seqreceiver	92	$seqreceiver_1$
seqreceiver	99	$seqreceiver_2$
seqreceiver	107	$seqreceiver_3$
datamsg	54	$datamsg_1$
datamsg	60	$datamsg_2$
msg	46	msg_1
msg	46	msg_2
pdu	108	pdu_1
pdu	101	pdu_2

ok	101	ok

Table 4. Definitions and uses of variables for the example ABP.

Node/edge		Node/edge	
3	$(d_{seqsender_1})_3$	(61,62)	$(p_{ok})_{(61,62)}$
4	$(d_{datamsg_1})_4$	62	$(c_{pdu_2})_{62}$, $(d_{datamsg_3})_{62}$
6	$(c_{datamsg_1})_6$, $(d_{datamsg_2})_6$	(63,64)	$(p_{ok})_{(63,64)}$
30	$(c_{seqsender_1})_{30}$, $(d_{seqsender_1})_{30}$	65	$(c_{datamsg_3})_{65}$
47	$(d_{seqsreceiver_1})_{47}$	66	$(c_{seqreceiver_1})_{66}$, $(d_{seqreceiver_1})_{66}$
49	$(c_{seqreceiver_1})_{49}$, $(d_{seqreceiver_2})_{49}$	67	$(c_{datamsg_2})_{67}$, $(d_{seqsender_2})_{67}$, $(d_{msg_1})_{67}$
51	$(c_{seqreceiver_2})_{51}$, $(d_{seqreceiver_3})_{51}$	68	$(c_{msg_2})_{68}$, $(d_{ack})_{68}$
(54,55)	$(p_{pdu_1})_{(54,55)}$ $(p_{seqreceiver_3})_{(54,55)}$	69	$(c_{msg_1})_{69}$, $(d_{pdu_1})_{69}$
56	$(c_{pdu_1})_{56}$, $(d_{pdu_2})_{56}$ $(d_{ok})_{56}$	70	$(c_{seqreceiver_3})_{70}$, $(d_{msg_2})_{70}$
(57,58)	$(p_{pdu_1})_{(57,58)}$ $(p_{seqreceiver_3})_{(57,58)}$	71	$(c_{seqreceiver_3})_{71}$, $(d_{msg_2})_{71}$
59	$(c_{pdu_1})_{59}$, $(d_{pdu_2})_{59}$ $(d_{ok})_{59}$		

4. Data Flow Analysis Based Validation Activities

The flowgraph representation of a given specification is intended for two main purposes, viz. data flow anomaly detection and data flow oriented test selection. By identifying definitions and uses of variables, the flowgraph representation of a specification provides the capability of detecting data flow anomalies which may indicate possible errors in the specification. Furthermore, by exposing the required information for establishing the associations between each use of a variable and those defs of variables which influence that use, the flowgraph representation provides the capability of applying data flow oriented test selection criteria to the specification. This results in the systematic selection of a set of test paths which can be used for validating data flow aspects of possible implementations for conformance to the specification.

4.1. Data Flow Anomaly Detection

The identification of definitions and uses of variables in a LOTOS specification facilitates data flow anomaly detection. A data flow anomaly is an abnormal pattern of actions on a variable during execution, abnormal in the sense that common expectations of how variables are to be used in the execution are violated. The occurrence of a data flow anomaly in a specification may be an indication of the existence of a specification error. Common errors such as misspelling, confusion among identifiers, and incorrect parameter usage often produce such anomalies. The detection of data flow anomalies can thus be used for detecting errors and for upgrading the quality of a specification, particularly of large and complex specifications.

Three important types of data flow anomalies can be distinguished, based on the classification of variable occurrences as definitions and uses [6]:

- using (c-use or p-use) a variable which is not defined before,
- redefining a variable before it is used, and
- defining a variable without subsequent use.

Anomalies of the first type, i.e. referencing undefined variables, would be static semantics errors in LOTOS. They can be detected prior to the static data flow analysis by a LOTOS static semantics analyzer [16]. Thus, only static semantically correct LOTOS specifications are considered to be mapped into a flowgraph representation. The second anomaly, redefining a variable without first using it, does not occur in LOTOS. Variables in LOTOS have a mathematical character and can only be bound to a value once; each definition of a variable introduces a new, unique variable identified by its name and the scope of its declaration. However, this particular declaration concept transforms an anomaly of the second type into an anomaly of the third type, i.e. defining a variable without subsequently using it. Anomalies of the third type typically do indicate possible errors in a LOTOS specification. Although it is syntactically and semantically allowed to have a definition of a variable without subsequent reference, this often refers to a badly written part in the specification. The following are some frequently occurring data flow anomalies in LOTOS specifications:

- a definition of a variable in the formal parameter list of a process definition does not have a subsequent use in the body of the process definition,
- a definition of a variable in the variable declaration list of an accept construct does not have a subsequent use in the enabled behaviour, or
- a definition of a variable without subsequent use occurs at a place where it would be appropriate to have a use of a previously defined variable.

A particular data flow anomaly of interest is in the ABP specification given in the appendix. This data flow anomaly is related to the variable 'ack' in line number 80. The variable is defined but never used (see Table 5). In fact, the analysis of this anomaly leads to the detection of a serious error. The acknowledgement received by the sender is never checked for its sequence number. This can lead to incorrect behaviour in the ABP, as shown by Fig. 3. The time sequence diagram in this figure illustrates a scenario in which the sender sends a message with sequence number 0. The message is received by the receiver, but before the corresponding acknowledgement reaches the sender, the timer expires, causing retransmission of the last message. After the sender receives the acknowledgement with sequence number 0, it sends a new message with sequence number 1, which gets lost. Meanwhile, the receiver receives the retransmitted message. This message is also acknowledged with sequence number 0, as the sender's sequence number and the receiver's sequence number do not match. If this acknowledgement is received by the sender before a timeout, it will be interpreted as an acknowledgement for the lost message and a new message may be sent by the sender.

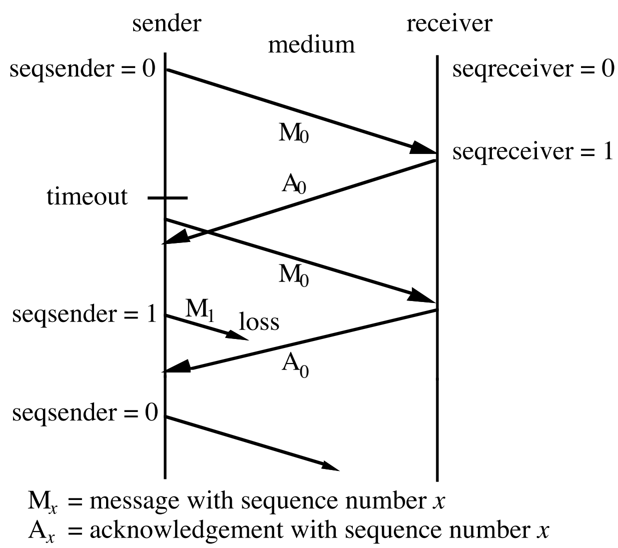


Fig. 3. Scenario of incorrect behaviour of the example ABP.

Clearly, the behaviour illustrated is incorrect; the message with sequence number 1 is lost and not retransmitted. This is due to the fact that the sequence number of the acknowledgement is not checked. The error can be overcome by specifying guards in which the sequence numbers of the sender and the sequence number of the acknowledgement are compared. This yields the following new process `messer` (see lines 79-81 of the Appendix):

```
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
```

In terms of definitions and uses the def of 'ack' now has two corresponding p-uses.

4.2 Data-flow Oriented Test Selection

The flowgraph representation of a behaviour expression facilitates the application of data flow oriented test selection criteria to the corresponding LOTOS specification by exposing the required information for establishing the associations between a def of a variable and those uses of variables that are influenced by that def. Several data flow oriented test selection criteria have been proposed for software testing purposes, among which the well-known ones are the *all-uses* [7, 20], *all-du-paths* [7, 20], *required k-tuples* [18], and *context coverage* [14] criteria. All of these criteria aim at selecting a set of paths that covers all occurrences of particular associations between definitions and uses in a given flowgraph representation of a system. A particular collection of test data should then cause the execution of an implementation under test along the selected set of paths, which allows testing the conformance of the data flow aspects of the implementation with respect to the intended system. The problem of how to select test data from the system's input domain is left outside the scope of this paper. The interested reader may refer to [7].

Each of the above criteria can be adapted for the flowgraph representation of a LOTOS specification. In this paper, only the adaptation of the all-uses criterion [20] is discussed. This requires revisiting some terminology [20].

A *path* $(n_1, n_2, ..., n_m)$ in a flowgraph $G(N, E)$ is a finite sequence of two or more nodes in N, such that for all i, $1 \le i \le m-1$, $(n_i, n_{i+1}) \in E$ $(m \ge 2)$. A path $(i_1, ..., i_k)$ is a *subpath* of path $(n_1, ..., n_m)$ if there exists a δ, $0 \le \delta \le m-k$, such that for all j, $1 \le j \le k$, $i_j = n_{j+\delta}$. A *path P_1 is covered by a path P_2* if P_1 is a subpath of P_2.

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}. The def of a variable x at a node i, $(\mathrm{d}_x)_i$, and the use of x at a node j or on an edge (j,k), $(\mathrm{u}_x)_j$ or $(\mathrm{u}_x)_{(j,k)}$ respectively, form a *def-use pair* (*du-pair*), denoted as $((\mathrm{d}_x)_i,(\mathrm{u}_x)_j)$ or $((\mathrm{d}_x)_i,(\mathrm{u}_x)_{(j,k)})$, if there exists a def-clear path wrt x from node i to node j or edge (j,k). For example, the def of the variable ok in *i*-node 59 and the p-use of ok on *r*-edge (61,62) form a du-pair $((\mathrm{d}_{\mathrm{ok}})_{59},(\mathrm{u}_{\mathrm{ok}})_{(61,62)})$. All du-pairs of the ABP specification in the appendix are given in Table 5. Note that reference is still made to the erroneous ABP specification. This suffices for illustration purposes, although it is common practice to first eliminate any specification errors before applying a test selection criterion.

Let P be a set of paths for the flowgraph of a given behaviour expression, then a du-pair $((d_x)_i,(u_x)_j)$ (or $((d_x)_i,(u_x)_{(j,k)})$) *is covered by P* if a def-clear path wrt x from node i to node j (or edge (j,k)) is a subpath of a path in P. Formally, a set of paths P satisfies the *all-uses* criterion in the flowgraph of a given behaviour expression if each du-pair in the flowgraph is covered by P. A set of test paths satisfying the all-uses criterion for the example ABP is given in Table 6.

Table 5. DU-pairs for the example ABP.

Variable	Def(s)	Use(s)
$seqsender_1$	$(d_{seqsender_1})_3$	$(c_{seqsender_1})_6$ $(c_{seqsender_1})_{30}$
	$(d_{seqsender_1})_{30}$	$(c_{seqsender_1})_6$ $(c_{seqsender_1})_{30}$
$seqsender_2$	$(d_{seqsender_2})_6$	$(c_{seqsender_2})_{67}$
$seqreceiver_1$	$(d_{seqreceiver_1})_{47}$	$(c_{seqreceiver_1})_{49}$ $(c_{seqreceiver_1})_{66}$
	$(d_{seqreceiver_1})_{66}$	$(c_{seqreceiver_1})_{49}$ $(c_{seqreceiver_1})_{66}$
$seqreceiver_2$	$(d_{seqreceiver_2})_{49}$	$(c_{seqreceiver_2})_{51}$
$seqreceiver_3$	$(d_{seqreceiver_3})_{51}$	$(c_{seqreceiver_3})_{70}$ $(c_{seqreceiver_3})_{71}$ $(p_{seqreceiver_3})_{(54,55)}$ $(p_{seqreceiver_3})_{(57,58)}$
$datamsg_1$	$(d_{datamsg_1})_4$	$(c_{datamsg_1})_6$
$datamsg_2$	$(d_{datamsg_2})_6$	$(c_{datamsg_2})_{67}$
$datamsg_3$	$(d_{datamsg_3})_{62}$	$(c_{datamsg_3})_{65}$
msg_1	$(d_{msg_1})_{67}$	$(c_{msg_1})_{69}$
msg_2	$(d_{msg_2})_{70}$ $(d_{msg_2})_{71}$	$(c_{msg_2})_{68}$ $(c_{msg_2})_{68}$
pdu_2	$(d_{pdu_2})_{56}$ $(d_{pdu_2})_{59}$	$(c_{pdu_2})_{62}$ $(c_{pdu_2})_{62}$
pdu_1	$(d_{pdu_1})_{69}$	$(c_{pdu_1})_{56}$ $(c_{pdu_1})_{59}$ $(p_{pdu_1})_{(54,55)}$ $(p_{pdu_1})_{(57,58)}$
ack	$(d_{ack})_{68}$	
ok	$(d_{ok})_{56}$	$(p_{ok})_{(61,62)}$ $(p_{ok})_{(63,64)}$
	$(d_{ok})_{59}$	$(p_{ok})_{(61,62)}$ $(p_{ok})_{(63,64)}$

5 Conclusions

We have shown that the explicit identification of the desired flow of data within the allowed control flow expressed by a system specification in LOTOS facilitates static data flow analysis based validation activities, namely, data flow anomaly detection and data flow oriented test selection. The information required for both these validation activities is explicitly provided by a flowgraph representation of the LOTOS specification.

Table 6. Test paths for the example ABP.

Test Path	du-pairs covered
(1,2,3,4,5,30,4,5,6)	$((d_{seqsender_1})_{3}, (u_{seqsender_1})_{30})$ $((d_{seqsender_1})_{30}, (u_{seqsender_1})_{6})$
(1,2,3,4,5,30,4,5,30)	$((d_{seqsender_1})_{3}, (u_{seqsender_1})_{30})$ $((d_{seqsender_1})_{30}, (u_{seqsender_1})_{30})$
(1,2,31,47,48,65,66,48,49)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{66})$ $((d_{seqreceiver_1})_{66}, (u_{seqreceiver_1})_{49})$
(1,2,31,47,48,65,66,48,65,66)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{66})$ $((d_{seqreceiver_1})_{66}, (u_{seqreceiver_1})_{66})$
(1,2,31,47,48,49,50,60,61,62,65)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{49})$ $((d_{datamsg_3})_{62}, (u_{datamsg_3})_{65})$
(1,2,3,4,5,6,7,8,9,10,67,34,35,38,69)	$((d_{seqsender_1})_{3}, (u_{seqsender_1})_{6})$ $((d_{seqsender_2})_{6}, (u_{seqsender_2})_{67})$ $((d_{datamsg_1})_{4}, (u_{datamsg_1})_{6})$ $((d_{datamsg_2})_{6}, (u_{datamsg_2})_{67})$ $((d_{msg_1})_{67}, (u_{msg_1})_{69})$
(1,2,31,47,48,49,50,51,52,53,54,55,70,41,42,45,68)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{49})$ $((d_{seqreceiver_2})_{49}, (u_{seqreceiver_2})_{51})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{(54,55)})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{70})$ $((d_{msg_2})_{70}, (u_{msg_2})_{68})$
(1,2,31,47,48,49,50,51,52,53,57,58,71,41,42,45,68)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{49})$ $((d_{seqreceiver_2})_{49}, (u_{seqreceiver_2})_{51})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{(57,58)})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{71})$ $((d_{msg_2})_{71}, (u_{msg_2})_{68})$
(1,2,31,47,48,49,50,51,52,53,54,55,56,60,61,62)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{49})$ $((d_{seqreceiver_2})_{49}, (u_{seqreceiver_2})_{51})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{(54,55)})$ $((d_{pdu_2})_{56}, (u_{pdu_2})_{62})$ $((d_{ok})_{56}, (u_{ok})_{(61,62)})$
(1,2,31,47,48,49,50,51,52,53,57,58,59,60,61,62)	$((d_{seqreceiver_1})_{47}, (u_{seqreceiver_1})_{49})$ $((d_{seqreceiver_2})_{49}, (u_{seqreceiver_2})_{51})$ $((d_{seqreceiver_3})_{51}, (u_{seqreceiver_3})_{(57,58)})$ $((d_{pdu_2})_{59}, (u_{pdu_2})_{62})$ $((d_{ok})_{59}, (u_{ok})_{(61,62)})$

(1,2,31,32,33,34,35,38,69,52,53,54,55,56)	$((d_{pdu_1})_{69}, (u_{pdu_1})_{(54,55)})$ $((d_{pdu_1})_{69}, (u_{pdu_1})_{56})$
(1,2,31,32,33,34,35,38,69,52,53,57,58,59)	$((d_{pdu_1})_{69}, (u_{pdu_1})_{(57,58)})$ $((d_{pdu_1})_{69}, (u_{pdu_1})_{59})$
(1,2,31,47,48,49,50,51,52,53,54,55,56,60,63,64)	$((d_{ok})_{56}, (u_{ok})_{(63,64)})$
(1,2,31,47,48,49,50,51,52,53,57,58,59,60,63,64)	$((d_{ok})_{59}, (u_{ok})_{(63,64)})$

By identifying definitions and uses of variables, the flowgraph representation of a specification provides the capability of detecting data flow anomalies which may indicate possible errors in the specification. For the ABP specification in the appendix we have shown that the analysis of such an anomaly can indeed lead to the detection of a serious error. It is important to detect specification errors at this early stage of the development process, since corrective actions for specification errors that are not detected until implementation and testing can be extremely costly.

By exposing the relative location of definitions and uses of variables within the allowed control flow in a specification, the corresponding flowgraph representation provides the capability of establishing the associations between each use of a variable and those defs of variables which influence that use, and subsequently facilitates the application of data flow oriented test selection criteria to the specification. We have shown the application of the all-uses criterion to select a set of test paths for testing the data flow aspects of an implementation of a given LOTOS specification.

The test paths selected to satisfy a particular test selection criterion within the flowgraph of a LOTOS specification correspond to particular traces of actions, which can be obtained from the flowgraph by extracting the labels of the *o*-nodes and *i*-nodes in these paths. However, some of the resulting traces may not be *executable* because of the existence of synchronization requirements. The traces must be embedded in executable action sequences, i.e. the test sequences, in which these requirements are fulfilled. Moreover, some of the originating test paths may not at all be *feasible* due to predicates on these paths which can never be satisfied. Unfortunately, the *static* nature of the analysis provides insufficient means to anticipate accurately the dynamic evolution of a system behaviour, a price which is in fact paid by any method relying on static information only [20]. It is, therefore, not always possible to algorithmically determine if a test path is feasible and/or if a corresponding trace of actions is executable. Yet, in an attempt to cope with the question of feasibility and executability, two approaches that are naturally based on the dynamic semantics of LOTOS are worth to consider. One of these approaches is the adaptation of the method employed in [7] which interactively selects a set of executable test paths. The other approach is to take into account a goal-oriented operational semantics for LOTOS [9] in an inference system to direct the derivation of executable test sequences.

Acknowledgements

This work was supported in part by the Telecommunications Research Institute of Ontario and the Natural Sciences and Engineering Research Council of Canada. We are grateful to our colleagues, E. Brinksma, L. Logrippo, M. Haj-Hussein and J. Sincennes, for many valuable discussions and suggestions.

References

[1] P. Berlinguette and D. Gueraichi, “The Alternating Bit Protocol in LOTOS: "Textual" and "Graphical" Representations,” TR 88-25, Dept. of Comput. Sci., University of Ottawa, Ottawa, Ontario, Canada, Aug. 1988.

[2] T. Bolognesi and E. Brinksma, “Introduction to the ISO specification language LOTOS,” *Comp. Networks ISDN Syst.*, vol. 14, 1987, pp. 25-59.

[3] E. Brinksma, “A Theory for the Derivation of Tests,” in *Protocol Specification, Testing, and Verification*, vol. 8, S. Aggarwal and K. Sabnani, Eds., Elsevier: North-Holland, 1988, pp. 63-74.

[4] A. Cavalli and Sung Un Kim, “Automated protocol conformance test generation based on formal methods for LOTOS specifications,” in *Proc. 5th Int. Workshop on Protocol Test Systems*, Montréal, Québec, Canada, 1992.

[5] T. Y. Cheung, Y. Wu, and X. Ye, “Generating test sequences and their degrees of indeterminism for protocols (with application to LOTOS),” in *Protocol Specification, Testing, and Verification*, vol. 11, B. Johnsson, J. Parrow, and B. Pehrson, Eds., Elsevier: North-Holland, 1991, pp. 278-293.

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

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

[8] 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., 1989, pp. 489-508.

[9] 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., 1992, pp. 305-320.

[10] M. S. Hecht, *Flow Analysis of Computer Programs*. New York: North-Holland, 1977.

[11] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] J. C. Huang, “Detection of data flow anomaly through program instrumentation,” *IEEE Trans. Softw. Eng.*, vol. SE-5, 1979, pp. 226-236.

[13] Open Systems Interconnection, "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, International Standard IS 8807, 1989.

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

[15] L. Logrippo, M. Faci, and M. Haj-Hussein, "An introduction to LOTOS: learning by examples," *Comp. Netw. ISDN Syst.*, vol. 23, 1992, pp. 325-342.

[16] L. Logrippo, A. Obaid, J.P. Briand, and M.C. Fehri, "An interpreter for LOTOS, a specification language for distributed systems," *Softw. - Pract. and Exper.*, vol.18, no. 4, 1988, pp. 365-385.

[17] R. Milner, *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[18] S. C. Ntafos, "On required element testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 11, 1984, pp. 795-803.

[19] L. J. Osterweil and L. D. Fosdick, "DAVE-A validation, error detection, and documentation system for Fortran programs," *Softw. – Pract. and Exper.*, vol. 6, 1976, pp. 473-486.

[20] S, Rapps and E.J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 4, 1985, pp. 367-375.

[21] J. H. Reif, "Data flow analysis of communicating processes," in *Proc. 6th Annual ACM Symp. Principles Progr. Lang.*, 1979.

[22] R. N. Taylor and L. J. Osterweil, "A facility for verification, testing, and documentation of concurrent process software," in *Proc. COMPSAC '78*, Chicago, IL, 1978.

[23] H. Ural, "Test sequence selection based on static data flow analysis," *Comp. Commun.*," vol. 10, no. 5, 1987, pp. 234-242.

[24] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing, *IEEE Trans. Commun.*," vol. COM-39, 1991, pp. 514-523.

[25] 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. Elsevier: North-Holland, 1989, pp. 145-158.

[26] M. R. Woodward, D. Hedley, and M. A. Hennel, "Experience with path analysis and testing of programs," *IEEE Trans. Softw. Eng.*, vol. SE-6, 1980, pp. 278-286.

APPENDIX

ALTERNATING BIT PROTOCOL SPECIFICATION IN LOTOS

```
specification alternate [sendgate, 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

hide sendpdu, receivepdu, sendack, receiveack in
  (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]
              >>
              tm  !reset; exit
            )
            [>
            ( tm  !expired; exit
               >>
               safemesser [sendpdu,receiveack] (datamsg, seqsender)
            )
         )
       )
       |[tm]|
       timer [tm]
     )

     where

          process messer [receiveack] : exit :=
          receiveack ?ack : message; exit
          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 *)
```