Test generation based on control and data dependencies within multi-process SDL specifications

Olaf Henniger[a], *Hasan Ural*[b]

[a] *GMD – German National Research Center for Information Technology*
Rheinstr. 75, 64295 Darmstadt, Germany
e-mail: henniger@darmstadt.gmd.de

[b] *School of Information Technology and Engineering*
University of Ottawa
Ottawa, Ontario, K1N 6N5, Canada
e-mail: ural@site.uottawa.ca

Abstract

Control and data flow aspects of a distributed system can be identified through the analysis of control and data dependencies that exist not only within processes, but also across process boundaries. This paper proposes a non-interleaving model that exposes the intra-process as well as inter-process control and data dependencies in a specification of a distributed system given in SDL. The model facilitates the generation of tests through the application of control-flow as well as data-flow oriented test selection criteria.

1 INTRODUCTION

In SDL [ITU92, Ell97], a distributed system is viewed as a collection of blocks and processes communicating with each other by exchanging signals through channels and signal routes. The externally observable behavior of a process is defined by an Extended Finite State Machine (EFSM). A specification in SDL expresses the desired control flow and data flow that must be established by a possible implementation of the specified system. The desired control flow is expressed as sequences of signals exchanged between processes. The desired data flow is expressed as relationships between the parameters associated with input signals, the local variables of processes, and the parameters associated with output signals.

Test generation from specifications in SDL has been widely studied. The existing methods for test generation can be roughly classified into methods with explicit test purposes and methods with implicit test purposes: methods with explicit test purposes require information

about the test purpose or the fault model for the generated test cases as input in addition to the specification; methods with implicit test purposes assume test purposes for the generated test cases implicitly and usually do not require supplementary inputs in addition to the specification.

The methods with explicit test purposes require the test designer to choose what to test and ensure that test cases consistent with the specification and the test purposes are generated. Most of the available test generation tools such as TGV [Fer96], SAMSTAG [Gra93], TVEDA [Gro97], Verilog's ObjectGeode [GEO96], Telelogic's Tau [TAU98], that are applicable to system specifications of a realistic size are based on methods with explicit test purposes. These methods offer much flexibility, but on the other hand they require considerable manual effort and do not guarantee a systematic test coverage.

For methods with implicit test purposes, the picture is reversed: While offering less flexibility in choosing what faults to generate test cases for, they require less manual efforts and guarantee a systematic test coverage. Some of these methods focus on the construction of test sequences for testing the control flow aspects. These methods abstract the control dependencies in the EFSM representation of a process as an FSM and apply FSM based test generation methods [Dah90, Sid89]. Other methods focus on the construction of test sequences for testing the data flow aspects [Sar87, Chun90, Ural91, Hen95, Ural00]. These methods identify the data dependencies in the EFSM representation of a process by applying principles of functional program testing [How87] or data flow analysis [Fos76]. Since these methods consider only a single EFSM and a limited SDL syntax for the EFSM representation, their applicability is restricted to a small subset of specifications in SDL. On the other hand, some methods with implicit test purposes have been proposed for systems of communicating EFSM's. As they need to explore the possible behavior of the system, these methods suffer from the state-explosion problem. Different approaches to alleviate the state-explosion problem have been proposed. [Lee93] pursues an approach similar to program slicing, pruning the given communicating FSM's to contain only a subset of actions, thus yielding a set of smaller, simplified specifications. [Ara91, Hen97] aim at diminishing the state explosion by generating noninterleaving models of the original specification by a reduced reachability analysis approach. TestComposer [Ker99] makes use of a reduced reachability analysis approach and implies as test purposes all transitions in the given SDL specification. Although this is a step in the right direction, the implied test purposes do not represent the set of functionalities in the given specification due to the fact that only an ordered set of individual transitions is a representation of a specific functionality of the system.

This paper proposes a model, called extended message flow graph (EMFG), exposing control and data dependencies not only within processes (intra-process dependencies), but also across process boundaries (inter-process dependencies) in a specification of a distributed system given in SDL. This model is intended for the generation of tests through the application of control-flow oriented as well as data-flow oriented test selection criteria [Mye79, Las83, Nta84, How87, Fra88] proposed in the literature for software testing. As studied in [Fra88, Har89] for block-structured programming languages such as Pascal, the application of each of these criteria requires the identification of control and/or data dependencies in a given program at intra-procedural or inter-procedural level. Analogously, for a system specification given in SDL as a collection of communicating processes, the proposed model facilitates the application of these criteria by exposing the intra-process dependencies within each process and the inter-process dependencies among communicating processes. The proposed extended

message flow graph and its construction rules are in part based on the adaptation of some earlier work for systems of asynchronously communicating state machines [Hen97] to specifications in SDL.

Section 2 introduces the extended message flow graph representation of a specification. An example is drawn from the Inres protocol specification [Ell97]. Section 3 deals with the generation of data-flow oriented tests from the extended message flow graph representation of a specification and adapts, as an example, the all-uses criterion to extended message flow graphs. Section 4 concludes the paper.

2 EXTENDED MESSAGE FLOW GRAPH OF A SPECIFICATION

2.1 Definitions

The proposed model for an SDL specification is an extended message flow graph (EMFG), based on message flow graphs (MFG) [Lad94]. Both MFG and EMFG are graphs representing concurrent processes exchanging messages. An MFG focuses on the communication behavior and control dependencies between processes and ignores pure computation statements inside processes, whereas an EMFG is capable of representing both control and data dependencies. An *extended message flow graph* (*EMFG*) is a triple $(N, \prec, \#)$ where

- N is a finite set of labeled nodes,
- $\prec \subseteq N \times N$ is an irreflexive flow relation, and
- $\# \subseteq N \times N$ is a symmetric conflict relation.

We distinguish the following types of nodes:

- send nodes (depicted as dots) representing outputs in the SDL specification,
- receive nodes (also depicted as dots) representing inputs in the SDL specification, and
- computation nodes (depicted as boxes) representing tasks and procedure calls in the SDL specification.

In the graphic representation of an EMFG, $n \prec n'$ is represented by a directed edge from node n to node n'. We distinguish the following types of directed edges:

- next-event edges (depicted as vertical or sloping arrows directed downwards) connecting nodes to their successors within the same process, and
- signal edges (depicted as horizontal or sloping arrows) connecting send nodes to receive nodes in other processes.

Next-event edges may be associated with boolean expressions representing decision predicates in the SDL specification. Parallel processes are represented with their next-event edges in parallel. The conflict relation is implicitly given in the graphic representation: Here, any two nodes n' and n'' within the same process that have the same predecessor node n, such that $n \prec n'$ and $n \prec n''$, are in conflict to each other, $n' \# n''$.

With respect to their graphic representation, EMFG's are closely related to message sequence charts (MSC) [ITU96]. The formal definition of EMFG's is closely related to that of flow event structures introduced in [Bou89]. Flow event structures are a generalization of prime event structures [Nie81] where the conflict between two events is not handed down to their successors, and the partial order relation of causality is replaced by an intransitive flow relation on events. Thus, an event can have different alternative enablings and flow event structures allow more compact descriptions of behavior.

We need the following definitions, which are closely related to the definitions for flow event structures. For a subset $C \subseteq N$, let $\prec_C$ be the restriction of the flow relation $\prec$ to C, and $\leq_C := \prec_C^*$ be the reflexive and transitive closure (i.e. a preorder) generated by $\prec_C$. A *configuration* C of an EMFG (N, $\prec$, #) is a finite subset of N such that:

- $\forall n, n' \in C: \neg(n \# n')$ (i.e., C is conflict-free),
- $n' \prec n \wedge n' \notin C \wedge n \in C \Rightarrow \exists\, n'' \in C: n' \# n'' \prec n$ (i.e., C is left-closed up to conflicts),
- the relation $\leq_C$ is an order relation (asymmetric, reflexive, and transitive relation) (i.e., C has no causality cycles).

Informally, a configuration of an EMFG is a partially ordered set of nodes of the EMFG that have been executed by some stage. The order of nodes is partial as only subsequent nodes within the same process and corresponding send and receive nodes are ordered, other nodes in different processes are concurrent and can be executed in more than one order. A configuration of an EMFG is a concept similar to a trace of a single state machine (a trace, however, is a totally ordered sequence) or to a configuration of an event structure (which is a partially ordered set of events).

A *path* $(n_1,n_2,...,n_m)$ in an EMFG is a sequence of nodes, such that $n_i \prec n_{i+1}$ for all i, $1 \leq i \leq m-1$, $m \geq 2$. A path $(n_1,n_2,...,n_m)$ is *covered* by a configuration C if $n_1,n_2,...,n_m \in C$. Let Π be a set of configurations of an EMFG. A path $(n_1,n_2,...,n_m)$ is *covered* by Π if Π contains a configuration C covering $(n_1,n_2,...,n_m)$.

2.2 Example

The construction of the EMFG of a specification is similar to reduced reachability analysis. We do not deal here with details of the construction algorithm, but present the EMFG for an example SDL specification. As an example, consider the well-known specification of the Inres protocol [Ell97]. The Inres protocol is used as demonstration example for many FDT based test generation methods [FMCT95]. It provides a simple data transfer service over an unreliable medium.

Figure 1 shows the EMFG for the initiator side of the Inres protocol. The initiator side consists of the block *Station_Ini* containing the two processes *Initiator* and *Coder*, which are here abbreviated as *I* and *C*. For easier orientation in the graphic representation, state nodes (depicted as ovals) have been added representing the states in the SDL specification. The repetition of a pair of state nodes stands for a loop back to the first occurrence of the pair in the graph. The labels of the nodes and the boolean expressions associated with edges of the EMFG are given in the first two columns of the Tables 1 and 2, respectively.

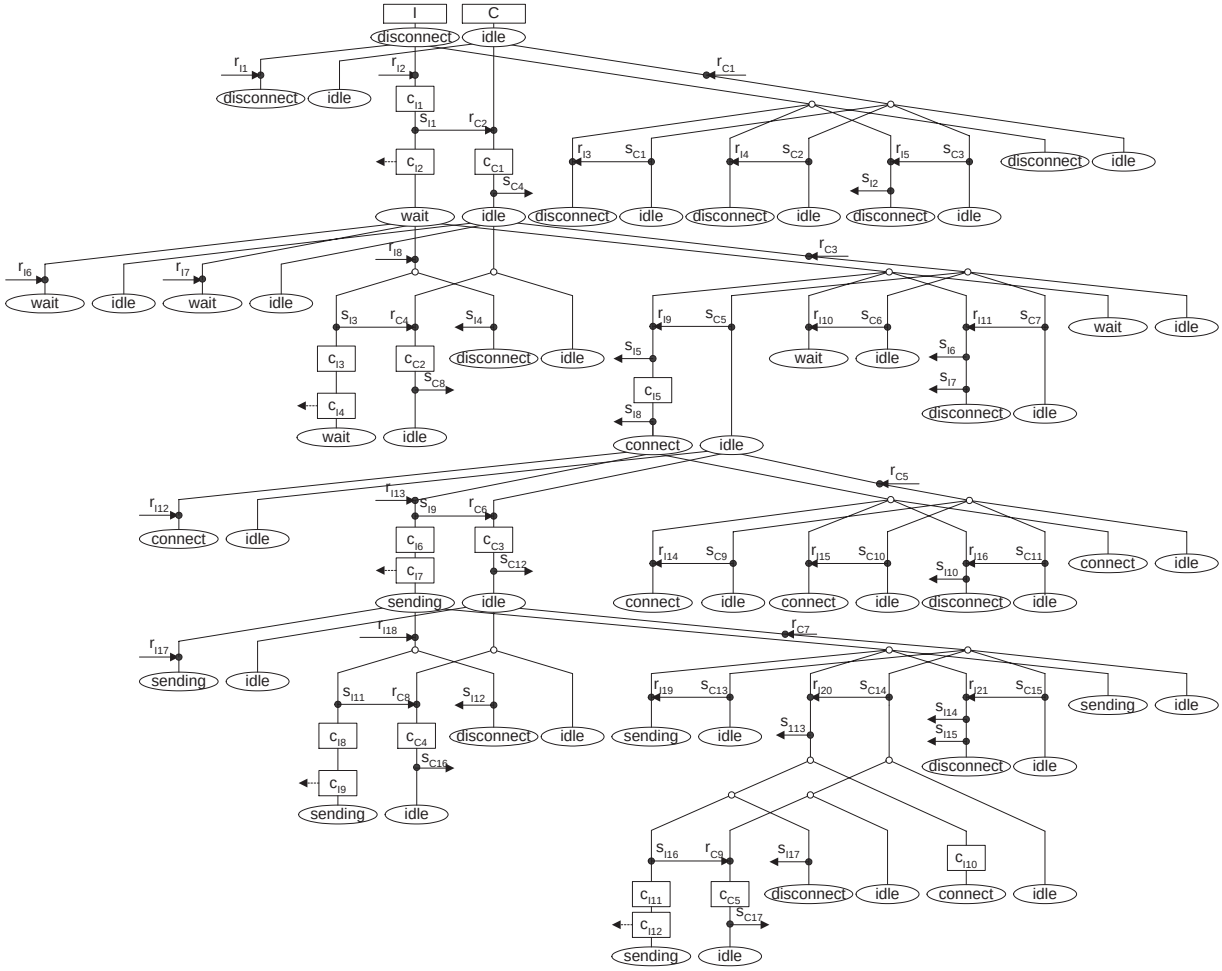

Figure 1 Extended message flow graph *G*.

Table 1 Nodes, statements, definitions, and c-uses in EMFG *G*.

Node	Statement	Definitions and c-uses
c_{C1}	sdu!id := CR	d(C.sdu!id)
c_{C2}	sdu!id := CR	d(C.sdu!id)
c_{C3}	sdu!id := DT, sdu!num:=num, sdu!data:=data	d(C.sdu!id), c(C.num), d(C.sdu!num), c(C.data), d(C.sdu!data)
c_{C4}	sdu!id := DT, sdu!num:=num, sdu!data:=data	d(C.sdu!id), c(C.num), d(C.sdu!num), c(C.data), d(C.sdu!data)
c_{C5}	sdu!id := DT, sdu!num:=num, sdu!data:=data	d(C.sdu!id), c(C.num), d(C.sdu!num), c(C.data), d(C.sdu!data)
c_{I1}	counter := 1	d(I.counter)
c_{I2}	set(now + 5, T)	
c_{I3}	counter := counter+1	c(I.counter), d(I.counter)
c_{I4}	set(now + 5, T)	
c_{I5}	number := 1	d(I.number)
c_{I6}	counter := 1	d(I.counter)
c_{I7}	set(now + 5, T)	
c_{I8}	counter := counter+1	c(I.counter), d(I.counter)
c_{I9}	set(now + 5, T)	
c_{I10}	number := succ(number)	c(I.number), d(I.number)
c_{I11}	counter := counter+1	c(I.counter), d(I.counter)
c_{I12}	set(now + 5, T)	
r_{C1}	input MDATind(sdu)	d(C.sdu!id),d(C.sdu!num)
r_{C2}	input CR	
r_{C3}	input MDATind(sdu)	d(C.sdu!id),d(C.sdu!num)
r_{C4}	input CR	
r_{C5}	input MDATind(sdu)	d(C.sdu!id),d(C.sdu!num)
r_{C6}	input DT(num, data)	c(DT.num), d(C.num), c(DT.data), d(C.data)
r_{C7}	input MDATind(sdu)	d(C.sdu!id),d(C.sdu!num)
r_{C8}	input DT(num, data)	c(DT.num), d(C.num), c(DT.data), d(C.data)
r_{C9}	input DT(num, data)	c(DT.num), d(C.num), c(DT.data), d(C.data)
r_{I1}	input IDATreq	
r_{I2}	input ICONreq	
r_{I3}	input CC	
r_{I4}	input AK(num)	c(AK.num), d(I.num)
r_{I5}	input DR	
r_{I6}	input IDATreq	

r_{I7}	input ICONreq	
r_{I8}	timeout T	
r_{I9}	input CC	
r_{I10}	input AK(num)	c(AK.num), d(I.num)
r_{I11}	input DR	
r_{I12}	input ICONreq	
r_{I13}	input IDATreq(data)	c(IDATreq.data),d(I.data)
r_{I14}	input CC	
r_{I15}	input AK(num)	c(AK.num), d(I.num)
r_{I16}	input DR	
r_{I17}	input ICONreq	
r_{I18}	timeout T	
r_{I19}	input CC	
r_{I20}	input AK(num)	c(AK.num), d(I.num)
r_{I21}	input DR	
s_{C1}	output CC	
s_{C2}	output AK(sdu!num)	c(C.sdu!num),d(AK.num)
s_{C3}	output DR	
s_{C4}	output MDATreq(sdu)	c(C.sdu!id)
s_{C5}	output CC	
s_{C6}	output AK(sdu!num)	c(C.sdu!num),d(AK.num)
s_{C7}	output DR	
s_{C8}	output MDATreq(sdu)	c(C.sdu!id)
s_{C9}	output CC	
s_{C10}	output AK(sdu!num)	c(C.sdu!num),d(AK.num)
s_{C11}	output DR	
s_{C12}	output MDATreq(sdu)	c(C.sdu!id), c(C.sdu!num), c(C.sdu!data)
s_{C13}	output CC	
s_{C14}	output AK(sdu!num)	c(C.sdu!num),d(AK.num)
s_{C15}	output DR	
s_{C16}	output MDATreq(sdu)	c(C.sdu!id), c(C.sdu!num), c(C.sdu!data)
s_{C17}	output MDATreq(sdu)	c(C.sdu!id), c(C.sdu!num), c(C.sdu!data)
s_{I1}	output CR	
s_{I2}	output IDISind	
s_{I3}	output CR	
s_{I4}	output IDISind	
s_{I5}	reset(T)	
s_{I6}	reset(T)	
s_{I7}	output IDISind	
s_{I8}	output ICONconf	
s_{I9}	output DT(number, data)	c(I.number), d(DT.num), c(I.data), d(DT.data)
s_{I10}	output IDISind	
s_{I11}	output DT(number, data)	c(I.number), d(DT.num), c(I.data), d(DT.data)
s_{I12}	output IDISind	
s_{I13}	reset(T)	
s_{I14}	reset(T)	
s_{I15}	output IDISind	
s_{I16}	output DT(number, data)	c(I.number), d(DT.num), c(I.data), d(DT.data)
s_{I17}	output IDISind	

Table 2 Edges, boolean expressions, and p-uses in EMFG *G*.

Edge	Boolean expression	P-uses
(r_{C1}, idle)	not (sdu!id = CC or sdu!id = AK or sdu!id = DR)	p(C.sdu!id)
(r_{C1}, s_{C1})	sdu!id = CC	p(C.sdu!id)
(r_{C1}, s_{C2})	sdu!id = AK	p(C.sdu!id)
(r_{C1}, s_{C3})	sdu!id = DR	p(C.sdu!id)
(r_{C3}, idle)	not (sdu!id = CC or sdu!id = AK or sdu!id = DR)	p(C.sdu!id)
(r_{C3}, s_{C5})	sdu!id = CC	p(C.sdu!id)
(r_{C3}, s_{C6})	sdu!id = AK	p(C.sdu!id)
(r_{C3}, s_{C7})	sdu!id = DR	p(C.sdu!id)
(r_{C5}, idle)	not (sdu!id = CC or sdu!id = AK or sdu!id = DR)	p(C.sdu!id)
(r_{C5}, s_{C9})	sdu!id = CC	p(C.sdu!id)
(r_{C5}, s_{C10})	sdu!id = AK	p(C.sdu!id)
(r_{C5}, s_{C11})	sdu!id = DR	p(C.sdu!id)
(r_{C7}, idle)	not (sdu!id = CC or sdu!id = AK or sdu!id = DR)	p(C.sdu!id)
(r_{C7}, s_{C13})	sdu!id = CC	p(C.sdu!id)
(r_{C7}, s_{C14})	sdu!id = AK	p(C.sdu!id)
(r_{C7}, s_{C15})	sdu!id = DR	p(C.sdu!id)
(r_{I8}, s_{I3})	counter < 4	p(I.counter)
(r_{I8}, s_{I4})	counter >= 4	p(I.counter)
(r_{I18}, s_{I11})	counter < 4	p(I.counter)
(r_{I18}, s_{I12})	counter >= 4	p(I.counter)
(s_{I13}, c_{I10})	num = number	p(I.num), p(I.number)
(s_{I13}, s_{I16})	not (num = number) and counter < 4	p(I.num), p(I.number), p(I.counter)
(s_{I13}, s_{I17})	not (num = number) and counter >= 4	p(I.num), p(I.number), p(I.counter)

3 DATA FLOW ORIENTED TEST SELECTION

3.1 Introduction

Data flow oriented test selection criteria allow the selective generation of test cases from a specification of the system under test. These criteria establish associations between definitions and uses of variables. Such associations are identified by tracking variables through the specification of the system, following them as they are modified, until they are ultimately used in outputs or to compute values for other variables. The criteria require that each of these associations is examined at least once during testing. The intuition behind the selection of tests based on the coverage of data flow associations is that faults in a system may lead to incorrect values and, as a result of propagation through computations, an error may show up at the system's output.

We will first define the data flow associations and the test selection criterion we are interested in, i.e. the all-uses criterion, and then present the application of the criterion to EMFG's representing specifications in SDL.

3.2 Classification of variable occurrences

Each variable occurrence in an EMFG is classified as being a definition, a computational use, or a predicate use which are referred to as *def*, *c-use*, and *p-use*, respectively. A *def* of variable *x* at node *n* (denoted by d_n^x) is an occurrence of *x* by which *x* gets a value. A *c-use* of variable *x* at node *n* (denoted by c_n^x) is an occurrence of *x* that directly affects the computation being performed (e.g., an occurrence of *x* on the right-hand side of an assignment statement) or allows one to see the result of some earlier defs (e.g., an occurrence of *x* in an output). A *p-use* of *x* on edge (*n*,*m*) (denoted by $p_{(n,m)}^x$) is an occurrence of *x* which directly affects the control flow (e.g., an occurrence of *x* in a boolean expression of a decision).

The following convention is used to classify each variable occurrence in an EMFG *G* as a def, c-use, or p-use:

a) input $s(X_1,...,X_n)$ in a receive node contains c-uses of the actual signal parameters followed by defs of the variables $X_1,...,X_n$; in the special case of an input from the environment, it contains only the defs of the variables $X_1,...,X_n$;
b) output $s(X_1,...,X_n)$ in a send node contains c-uses of the variables $X_1,...,X_n$ followed by defs of the actual signal parameters; in the special case of an output to the environment, it contains only the c-uses of the variables $X_1,...,X_n$;
c) an assignment statement Y := *expression*[1] in a computation node contains c-uses of all variables occurring in the expression followed by a def of the variable Y;
d) a boolean expression on a next-event edge contains p-uses of all variables occurring in the expression;

[1] An *expression* is either a constant or an *n*-ary function $f(Y_1,...,Y_n)$, $n \geq 1$, where $Y_1,...,Y_n$ are variables.

e) a procedure call[2] $p_i(X_1,...,X_m,e_{m+1},...,e_n)$ contains a c-use of each variable X_i ($1 \le i \le m$) and a c-use of each variable Y_j occurring in an expression e_k ($m+1 \le k \le n$), followed by a def of each X_i.

The classification of the variables occurring in the parameter list of a procedure call is based on the required accuracy of data flow representation in the specification. For the purposes of this paper we follow the classification of [Fra88] which is sufficient for criteria based on individual du-pairs.

For the example in Figure 1, the classification of variable occurrences as definitions, computational uses, or predicate uses is shown in the third column of Table 1 and 2.

3.3 Data flow associations

The identification of defs, c-uses, and p-uses of variables in an EMFG facilitates tracing the flow of data and establishing data flow associations among occurrences of variables.

A path $(n_1,n_2,...,n_{r-1},n_r)$ in an EMFG G is said to be a *def-clear path* with respect to a variable x from node n_1 to node n_r or from node n_1 to edge (n_{r-1},n_r) if either $r = 2$, or $r > 2$ and there are no definitions of x at nodes n_2 to n_{r-1}. A definition d_i^x and a c-use c_j^x form a *du-pair* (represented by the tuple (d_i^x, c_j^x)) if there is a def-clear path with respect to x from node i to node j. Similarly, d_i^x and $p_{(j,k)}^x$ form a du-pair (represented by the tuple $(d_i^x, p_{(j,k)}^x)$) if there is a def-clear path with respect to x from node i to node j.

Table 3 du-Pairs in the EMFG *G*.

No.	Def	Use	Shortest def-clear path
1	$d_{c_{C1}}^{C.sdu!id}$	$c_{s_{C4}}^{C.sdu!id}$	c_{C1}, s_{C4}
2	$d_{c_{C2}}^{C.sdu!id}$	$c_{s_{C8}}^{C.sdu!id}$	c_{C2}, s_{C8}
3	$d_{c_{C3}}^{C.sdu!id}$	$c_{s_{C12}}^{C.sdu!id}$	c_{C3}, s_{C12}
4	$d_{c_{C3}}^{C.sdu!num}$	$c_{s_{C12}}^{C.sdu!num}$	c_{C3}, s_{C12}
5	$d_{c_{C3}}^{C.sdu!data}$	$c_{s_{C12}}^{C.sdu!data}$	c_{C3}, s_{C12}
6	$d_{c_{C4}}^{C.sdu!id}$	$c_{s_{C16}}^{C.sdu!id}$	c_{C4}, s_{C16}
7	$d_{c_{C4}}^{C.sdu!num}$	$c_{s_{C16}}^{C.sdu!num}$	c_{C4}, s_{C16}
8	$d_{c_{C4}}^{C.sdu!data}$	$c_{s_{C16}}^{C.sdu!data}$	c_{C4}, s_{C16}
9	$d_{c_{C5}}^{C.sdu!id}$	$c_{s_{C17}}^{C.sdu!id}$	c_{C5}, s_{C17}
10	$d_{c_{I1}}^{I.counter}$	$p_{(r_{I8},s_{I3})}^{I.counter}$	c_{I1}, s_{I1}, c_{I2}, r_{I8}, s_{I3}
11	$d_{c_{I1}}^{I.counter}$	$c_{c_{I3}}^{I.counter}$	c_{I1}, s_{I1}, c_{I2}, r_{I8},s_{I3},c_{I3}
12	$d_{c_{I1}}^{I.counter}$	$p_{(r_{I8},s_{I4})}^{I.counter}$	c_{I1}, s_{I1}, c_{I2}, r_{I8}, s_{I4}
13	$d_{c_{I10}}^{I.number}$	$p_{(s_{I13},c_{I10})}^{I.number}$	c_{I10}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, c_{I10}
14	$d_{c_{I10}}^{I.number}$	$p_{(s_{I13},s_{I16})}^{I.number}$	c_{I10}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I16}
15	$d_{c_{I10}}^{I.number}$	$p_{(s_{I13},s_{I17})}^{I.number}$	c_{I10}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I17}
16	$d_{c_{I11}}^{I.counter}$	$p_{(r_{I18},s_{I11})}^{I.counter}$	c_{I11}, c_{I12}, r_{I18}, s_{I11}
17	$d_{c_{I11}}^{I.counter}$	$p_{(r_{I18},s_{I12})}^{I.counter}$	c_{I11}, c_{I12}, r_{I18}, s_{I12}
18	$d_{c_{I11}}^{I.counter}$	$p_{(s_{I13},s_{I16})}^{I.counter}$	c_{I11}, c_{I12}, r_{I20},s_{I13},s_{I16}
19	$d_{c_{I11}}^{I.counter}$	$p_{(s_{I13},s_{I17})}^{I.counter}$	c_{I11}, c_{I12}, r_{I20},s_{I13},s_{I17}

[2] A *procedure call* is in the form $p_i(X_1,...,X_m,e_{m+1},...,e_n)$ where p_i is the procedure identifier, $X_1,...,X_m$ are variables representing actual *in/out parameters*, and $e_{m+1},...,e_n$ are expressions representing actual *in parameters* [Ural91].

20	$d^{I.counter}_{c_{I3}}$	$p^{I.counter}_{(r_{I8}, s_{I3})}$	c_{I3}, c_{I4}, r_{I8}, s_{I3}
21	$d^{I.counter}_{c_{I3}}$	$p^{I.counter}_{(r_{I8}, s_{I4})}$	c_{I3}, c_{I4}, r_{I8}, s_{I4}
22	$d^{I.number}_{c_{I5}}$	$c^{I.number}_{s_{I9}}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}
23	$d^{I.number}_{c_{I5}}$	$c^{I.number}_{s_{I11}}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I18}, s_{I11}
24	$d^{I.number}_{c_{I5}}$	$c^{I.number}_{c_{I10}}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, c_{I10}
25	$d^{I.number}_{c_{I5}}$	$p^{I.number}_{(s_{I13}, c_{I10})}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, c_{I10}
26	$d^{I.number}_{c_{I5}}$	$p^{I.number}_{(s_{I13}, s_{I16})}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I16}
27	$d^{I.number}_{c_{I5}}$	$p^{I.number}_{(s_{I13}, s_{I17})}$	c_{I5}, s_{I8}, r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I17}
28	$d^{I.counter}_{c_{I6}}$	$p^{I.counter}_{(r_{I18}, s_{I11})}$	c_{I6}, c_{I7}, r_{I18}, s_{I11}
29	$d^{I.counter}_{c_{I6}}$	$c^{I.counter}_{c_{I8}}$	c_{I6}, c_{I7}, r_{I18}, s_{I11}, c_{I8}
30	$d^{I.counter}_{c_{I6}}$	$p^{I.counter}_{(r_{I18}, s_{I12})}$	c_{I6}, c_{I7}, r_{I18}, s_{I12}
31	$d^{I.counter}_{c_{I6}}$	$p^{I.counter}_{(s_{I13}, s_{I16})}$	c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I16}
32	$d^{I.counter}_{c_{I6}}$	$c^{I.counter}_{c_{I11}}$	c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I16}, c_{I11}
33	$d^{I.counter}_{c_{I6}}$	$p^{I.counter}_{(s_{I13}, s_{I17})}$	c_{I6}, c_{I7}, r_{I20}, s_{I13}, s_{I17}
34	$d^{I.counter}_{c_{I8}}$	$p^{I.counter}_{(r_{I18}, s_{I11})}$	c_{I8}, c_{I9}, r_{I18}, s_{I11}
35	$d^{I.counter}_{c_{I8}}$	$p^{I.counter}_{(r_{I18}, s_{I12})}$	c_{I8}, c_{I9}, r_{I18}, s_{I12}
36	$d^{C.sdu!id}_{r_{C1}}$	$p^{C.sdu!id}_{(r_{C1}, idle)}$	r_{C1}, idle
37	$d^{C.sdu!id}_{r_{C1}}$	$p^{C.sdu!id}_{(r_{C1}, s_{C1})}$	r_{C1}, s_{C1}
38	$d^{C.sdu!id}_{r_{C1}}$	$p^{C.sdu!id}_{(r_{C1}, s_{C2})}$	r_{C1}, s_{C2}
39	$d^{C.sdu!id}_{r_{C1}}$	$p^{C.sdu!id}_{(r_{C1}, s_{C3})}$	r_{C1}, s_{C3}
40	$d^{C.sdu!id}_{r_{C3}}$	$p^{C.sdu!id}_{(r_{C3}, idle)}$	r_{C3}, idle
41	$d^{C.sdu!id}_{r_{C3}}$	$p^{C.sdu!id}_{(r_{C3}, s_{C5})}$	r_{C3}, s_{C5}
42	$d^{C.sdu!id}_{r_{C3}}$	$p^{C.sdu!id}_{(r_{C3}, s_{C6})}$	r_{C3}, s_{C6}
43	$d^{C.sdu!id}_{r_{C3}}$	$p^{C.sdu!id}_{(r_{C3}, s_{C7})}$	r_{C3}, s_{C7}
44	$d^{C.sdu!id}_{r_{C5}}$	$p^{C.sdu!id}_{(r_{C5}, idle)}$	r_{C5}, idle
45	$d^{C.sdu!id}_{r_{C5}}$	$p^{C.sdu!id}_{(r_{C5}, s_{C10})}$	r_{C5}, s_{C10}
46	$d^{C.sdu!id}_{r_{C5}}$	$p^{C.sdu!id}_{(r_{C5}, s_{C11})}$	r_{C5}, s_{C11}
47	$d^{C.sdu!id}_{r_{C5}}$	$p^{C.sdu!id}_{(r_{C5}, s_{C9})}$	r_{C5}, s_{C9}
48	$d^{C.num}_{r_{C6}}$	$c^{C.num}_{c_{C3}}$	r_{C6}, c_{C3}
49	$d^{C.data}_{r_{C6}}$	$c^{C.data}_{c_{C3}}$	r_{C6}, c_{C3}
50	$d^{C.sdu!id}_{r_{C7}}$	$p^{C.sdu!id}_{(r_{C7}, idle)}$	r_{C7}, idle
51	$d^{C.sdu!id}_{r_{C7}}$	$p^{C.sdu!id}_{(r_{C7}, s_{C13})}$	r_{C7}, s_{C13}
52	$d^{C.sdu!id}_{r_{C7}}$	$p^{C.sdu!id}_{(r_{C7}, s_{C14})}$	r_{C7}, s_{C14}
53	$d^{C.sdu!id}_{r_{C7}}$	$p^{C.sdu!id}_{(r_{C1}, s_{C15})}$	r_{C7}, s_{C15}
54	$d^{C.num}_{r_{C8}}$	$c^{C.num}_{c_{C4}}$	r_{C8}, c_{C4}
55	$d^{C.data}_{r_{C8}}$	$c^{C.data}_{c_{C4}}$	r_{C8}, c_{C4}
56	$d^{I.data}_{r_{I13}}$	$c^{I.data}_{s_{I9}}$	r_{I13}, s_{I9}
57	$d^{I.data}_{r_{I13}}$	$c^{I.data}_{s_{I11}}$	r_{I13}, s_{I9}, c_{I6}, c_{I7}, r_{I18}, s_{I11}
58	$d^{AK.num}_{s_{C10}}$	$c^{AK.num}_{r_{I15}}$	s_{C10}, r_{I15}
59	$d^{AK.num}_{s_{C14}}$	$c^{AK.num}_{r_{I20}}$	s_{C14}, r_{I20}
60	$d^{AK.num}_{s_{C2}}$	$c^{AK.num}_{r_{I4}}$	s_{C2}, r_{I4}
61	$d^{AK.num}_{s_{C6}}$	$c^{AK.num}_{r_{I10}}$	s_{C6}, r_{I10}
62	$d^{DT.num}_{s_{I11}}$	$c^{DT.num}_{r_{C8}}$	s_{I11}, r_{C8}
63	$d^{DT.data}_{s_{I11}}$	$c^{DT.data}_{r_{C8}}$	s_{I11}, r_{C8}
64	$d^{DT.num}_{s_{I16}}$	$c^{DT.num}_{r_{C9}}$	s_{I16}, r_{C9}
65	$d^{DT.data}_{s_{I16}}$	$c^{DT.data}_{r_{C9}}$	s_{I16}, r_{C9}
66	$d^{DT.num}_{s_{I9}}$	$c^{DT.num}_{r_{C6}}$	s_{I9}, r_{C6}
67	$d^{DT.data}_{s_{I9}}$	$c^{DT.data}_{r_{C6}}$	s_{I9}, r_{C6}

3.4 All-uses criterion

Based on the definition of a du-pair, a variety of data-flow oriented test generation criteria have been proposed [Las83, Nta84, Fra88]. In this paper we consider the all-uses criterion [Fra88] for illustrating the use of the EMFG model.

The all-uses criterion requires that every du-pair in a given EMFG be covered at least once during testing. In terms of the EMFG model this means that a set Π of configurations of an EMFG G is to be selected covering each du-pair in G at least once. A set Π of configurations of an EMFG G is said to *cover* a du-pair in G if Π covers a def-clear path for that du-pair. Formally, a set Π of configurations satisfies the *all-uses criterion* for an EMFG G if and only if every du-pair in G is covered at least once by Π.

The result of the application of the all-uses criterion to the EMFG G in Figure 1 is shown in Table 4. The smallest set of configurations that covers the shortest def-clear paths of all du-pairs has been selected. To improve readability, subsets of related nodes are enclosed in parentheses.

Table 4 Set of configurations satisfying the all-uses criterion for EMFG G.

No.	Configuration
1	(r_{C1}, idle)
2	(r_{C1}, s_{C1}), (r_{I3})
3	(r_{C1}, s_{C2}), (r_{I4})
4	(r_{C1}, s_{C3}), (r_{I5}, s_{I2})
5	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, idle)
6	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9},s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{I18}, s_{I11}, c_{I8}, c_{I9}), (r_{C8}, c_{C4}, s_{C16}), (r_{I18}, s_{I11}, c_{I8}, c_{I9}), (r_{C8}, c_{C4}, s_{C16})
7	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{I18}, s_{I11}, c_{I8}, c_{I9}), (r_{C8}, c_{C4}, s_{C16}), (r_{I18}, s_{I12})
8	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{I18}, s_{I12})
9	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, c_{I10}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, c_{I10})
10	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, c_{I10}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17})
11	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, c_{I10}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, s_{I17})
12	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17}), (r_{I18}, s_{I11}, c_{I8}, c_{I9}), (r_{C8}, c_{C4}, s_{C16})
13	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17}), (r_{I18}, s_{I12})
14	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17})
15	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, s_{I16}, c_{I11}, c_{I12}), (r_{C9}, c_{C5}, s_{C17}), (r_{C7}, s_{C14}), (r_{I20}, s_{I13}, s_{I17})
16	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C14}), (r_{I20}, s_{I13}, s_{I17})
17	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}), (r_{C7},idle)
18	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C13}), (r_{I19})
19	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{I13}, s_{I9}, c_{I6}, c_{I7}), (r_{C6}, c_{C3}, s_{C12}),(r_{C7},s_{C15}), (r_{I21}, s_{I14}, s_{I15})
20	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{C5}, idle)
21	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{C5}, s_{C9}), (r_{I14})
22	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{C5}, s_{C10}), (r_{I15})
23	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C5}), (r_{I9}, s_{I5}, c_{I5}, s_{I8}), (r_{C5}, s_{C11}), (r_{I16}, s_{I10})
24	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C6}), (r_{I10})
25	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{C3}, s_{C7}), (r_{I11}, s_{I6}, s_{I7})
26	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8})
27	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8}), (r_{I8}, s_{I4})
28	(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{I8}, s_{I4})

Each configuration in a set Π of configurations is associated with a boolean expression called *feasibility predicate* which is a conjunction of all the boolean expressions occurring on

the next-event edges within the configuration. A feasibility predicate for a configuration can be represented as a system of equalities and inequalities. If this system has a solution, then the corresponding configuration is called *feasible*. Otherwise, the configuration is called *infeasible*. It can be shown, by using decidability theory, that the selection of infeasible configurations cannot always be avoided. Thus, in the general case, one has to determine the feasibility of a selected configuration manually, i.e., the corresponding feasibility predicate must be formed by using symbolic execution [Mye79] and a solution to the system of inequalities representing the feasibility predicate must be sought.

For the particular case of this example, not all configurations in Table 4 are feasible. The configurations 7, 8, 11, 13, 15, 16, 27, and 28 are infeasible as the predicate *counter* ≥ 4 occurring on next-event edges is not satisfied within these configurations. The remaining, feasible configurations in Table 4 do not cover the du-pairs 12, 15, 17, 19, 21, 27, 30, 33, and 35. Some of these du-pairs, namely 15, 17, 19, 21, 27, and 35, can be covered by feasible configurations that are, however, not in the set of smallest configurations given in Table 4. For instance, the du-pair 21 ($d_{c_{I3}}^{I.counter}$, $p_{(r_{I8},s_{I4})}^{I.counter}$), covered by the infeasible configuration 27, can be covered by the feasible configuration {(r_{I2}, c_{I1}, s_{I1}, c_{I2}), (r_{C2}, c_{C1}, s_{C4}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8}), (r_{I8}, s_{I3}, c_{I3}, c_{I4}), (r_{C4}, c_{C2}, s_{C8}), (r_{I8}, s_{I4})}, which traverses the loop until *counter* = 4. For the remaining du-pairs, namely 12, 30, and 33, there is no feasible configuration. For instance, the du-pair 12 ($d_{c_{I1}}^{I.counter}$, $p_{(r_{I8},s_{I4})}^{I.counter}$), covered by the infeasible configuration 28, cannot be covered by any feasible configuration: Covering this du-pair requires to cover a def-clear path with respect to *counter* from the def of *counter* in *counter* := 1 to its p-use in *counter* ≥ 4. Therefore, in any configuration covering this du-pair the predicate *counter* ≥ 4 is not satisfied.

3.5 Mapping of selected configurations to test cases

We assume that the tester is, in general, distributed into a main test component and several parallel test components and that each process of the distributed system is observed by a separate test component. Then, each configuration in the set of configurations satisfying the all-uses criterion for an EMFG can easily be mapped to a test case description: Following the next-event edges between the nodes within a configuration, the externally visible signals are inverted (i.e. inputs of the specified system are mapped to outputs of the tester and vice-versa) and recorded in separate behavior trees for each test component. In the special case that one wants signals from different concurrent processes to be observed by the same test component, the interleavings (i.e. all possible sequences) of these signals have to be computed. The values of signal parameters are determined by symbolic execution along the selected configuration.

At the beginning of the behavior description of the main test component, CREATE constructs are inserted for activating the parallel test components. If a signal of a test component TC_i is immediately succeeded by a signal of another test component TC_k, then a coordination message from TC_i to TC_k is inserted into the test case description to inform TC_k about the occurrence of the signal in TC_i. Conflicting inputs to the tester that are permitted by the specification, but are outside the selected configuration have to be taken into account in the test case description as alternatives leading to INCONCLUSIVE verdicts. At the end of the selected configuration, a PASS verdict is assigned. Finally, OTHERWISE events, leading to

FAIL verdicts, have to be added to each level of indentation containing receive events to deal with any unexpected behavior.

The test architecture for the example, the initiator side of the Inres protocol, includes two test components, a main test component at the upper interface of the process *I* (*Initiator*) and a parallel test component at the lower interface of the process *C* (*Coder*). Table 5 sketches the test case corresponding to configuration 9 in Table 4 in (relaxed) Concurrent TTCN notation [Bau94].

Table 5 Test case corresponding to configuration 9 in Table 5.

```
CREATE(Medium : MediumTree)
    U!ICONreq
        START T(5)
            U?ICONconf
                CANCEL T
                    U!IDATreq(data1)
                        START T(5)
                            CP1?CM1
                                CANCEL T
                                    U!IDATreq(data2)
                                        START T(5)
                                            CP1?CM1                 PASS
                                                CANCEL T
                                            ?TIMEOUT T              INCONC
                            ?TIMEOUT T                              INCONC
            ?TIMEOUT T                                              INCONC
MediumTree
M?MDATreq((.CR, *, *.))
    M!MDATind((.CC, *, *.))
        M?MDATreq((.DT, 1, data1.))
            M!MDATind((.AK, 1, *.))
                CP1!CM1
                    M?MDATreq((.DT, 0, data2.))
                        M!MDATind((.AK, 0, *.))
                            CP1!CM1                                 (PASS)
```

4 CONCLUSIONS

We have presented extended message flow graphs as a non-interleaving model that exposes control and data dependencies not only within processes, but also across process boundaries in a SDL specification of a distributed system. The EMFG model is intended for the generation of tests through the application of control-flow oriented as well as data-flow oriented test selection criteria.

As an example, the data-flow oriented all-uses criterion has been adapted to the EMFG model. Its application has been successfully demonstrated for an example specification of a distributed system.

REFERENCES

[Ara91] N. Arakawa and T. Soneoka, “A test case generation method for concurrent programs”, in Proc. of IWPTS’91, Leidschendam, The Netherlands, 1991.

[Bau94] B. Baumgarten and A. Giessler, *OSI conformance testing methodology and TTCN*, North-Holland, 1994.

[Bou89] G. Boudol and I. Castellani, “Permutation of transitions: An event structure semantics for CCS and SCCS”, in J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.), *Linear time, branching time, and partial order in logics and models for concurrency*, Lecture Notes in Computer Science, vol. 354, Springer, 1989, pp. 411–427.

[Chun90] W. Chun and P.D. Amer, “Test case generation for protocols specified in Estelle”, in Proc. of FORTE’90, Madrid, Spain, 1990, pp. 197–210.

[Dah90] A.T. Dahbura, K.K. Sabnani, and M.U. Uyar, “Formal methods for generating protocol conformance test sequences”, *Proceedings of the IEEE*, 78, 8, 1990, pp. 1317–1325.

[Ell97] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL – formal object-oriented language for communicating systems*, Prentice Hall, 1997.

[Fer96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho, “Using on-the-fly verification techniques for the generation of test suites”, in Proc. of CAV’96, New Brunswick, NJ, USA, 1996, pp. 348–359.

[FMCT95] FMCT guidelines on test generation methods from formal descriptions, ITU-T Q.8/10 and ISO/JTC1/SC21/Project 54.2, 1995.

[Fos76] L.D. Fosdick and L.J. Osterweil, “Data flow analysis in software reliability”, *ACM Computing Surveys*, 8, 3, 1976, pp. 305–330.

[Fra88] P.G. Frankl and E.J. Weyuker, “An applicable family of data flow testing criteria”, *IEEE Trans. Software Eng.*, 14, 10, 1988, pp. 1483–1498.

[GEO96] ObjectGeode, Verilog, Toulouse, France, 1996.

[Gra93] J. Grabowski, D. Hogrefe, and R. Nahm, “Test case generation with test purpose specification by MSCs”, in Proc. of the 6th SDL Forum, Darmstadt, Germany, 1993.

[Gro97] R. Groz and N. Risser, “Eight years of experience in test generation from FDTs using TVEDA”, in Proc. of FORTE/PSTV’97, Chapman and Hall, 1997.

[Har89] M.J. Harrold and M.L. Soffa, “Interprocedural data flow testing”, in Proc. of 3rd Symposium on Testing, Analysis, and Verification, Key West, Florida, 1989, pp. 158–167.

[Hen95] O. Henniger, A. Ulrich, and H. König, “Transformation of Estelle modules aiming at test case generation”, in Proc. of IWPTS’95, Evry, France, 1995.

[Hen97] O. Henniger, “On test case generation from asynchronously communicating state machines”, in Proc. of IWTCS’97, Cheju Island, South Korea, 1997.

[How87] W.E. Howden, *Functional program testing and analysis*, McGraw Hill, New York, 1987.

[ITU92] Specification and Description Language (SDL), ITU-T Recommendation Z.100, 1992.

[ITU96] Message Sequence Chart (MSC), ITU-T Recommendation Z.120, 1996.

[Ker99] A. Kerbrat, T. Jeron, and R. Groz, “Automated test generation from SDL specifications”, in Proc. of SDL Forum ’99, Montreal, Canada, 1999, pp. 135–151.

[Kor87] B. Korel, “The program dependence graph in static program testing”, *Info. Processing Letters*. 24, 1987, pp. 103–108.

[Lad94] P.B. Ladkin and S. Leue, “Interpreting message flow graphs”, *Formal Aspects of Computing*, 1994.

[Las83] J.W. Laski and B. Korel, “A data-flow oriented program testing strategy”, *IEEE Trans. Software Eng.*, vol. SE-9, no. 5, 1983, pp. 347–354.

[Lee93] D. Lee, K.K. Sabnani, D.M. Kristol, S. Paul, “Conformance testing of protocols specified as communicating FSMs”, in Proc. of IEEE INFOCOM’93, San Francisco, CA, USA, 1993.

[Mye79] G.J. Myers, *The art of software testing.* New York: John Wiley & Sons, 1979.

[Nie81] M. Nielsen, G. Plotkin, and G. Winskel, “Petri nets, event structures and domains, part I”, *Theoretical Computer Science*, vol. 13, 1985, pp. 85-108.

[Nta84] S.C. Ntafos, “On required element testing”, *IEEE Trans. Software Eng.,* vol. SE-10, no. 11, 1984, pp. 795–803.

[Sar87] B. Sarikaya, G.v. Bochmann, and E. Cerny, “A test design methodology for protocol testing”, *IEEE Trans. Software Eng.*, vol. SE-13, no. 5, 1987, pp. 518–531.

[Sid89] D.P. Sidhu and T.K. Leung, “Formal methods for protocol testing: A detailed study”, *IEEE Trans. Software Eng.*, vol. SE-15, no. 4, 1989, pp. 413–426.

[TAU98] Telelogic Tau 3.4, Telelogic AB, Malmo, Sweden, 1998.

[Ural91] H. Ural and B. Yang, “A test sequence generation method for protocol testing”, *IEEE Trans. Commun.*, 39, 4, 1991, pp. 514–523.

[Ural00] H. Ural, K. Saleh, A. Williams, “Test generation based on control and data dependencies within system specifications in SDL”, *Computer Communications*, 23, 2000, pp. 609–627.