

# On The Testability of SDL Specifications

R. M. Hierons<sup>a</sup> T.-H. Kim<sup>b</sup> H. Ural<sup>c</sup>

<sup>a</sup>*Department of Information Systems and Computing, Brunel University,  
Uxbridge, Middlesex, UB8 3PH, United Kingdom*

<sup>b</sup>*School of Computer and Software Engineering, Kumoh National Institute of  
Technology, 188 Sinpyeong-dong, Gumi-si, Gyeongbuk 730-701, Korea*

<sup>c</sup>*School of Information Technology and Engineering, Faculty of Engineering,  
University of Ottawa, 800 King Edward Avenue, Ottawa, Ontario, K1N 6N5,  
Canada*

---

## Abstract

The problem of testing from an SDL specification is often complicated by the presence of infeasible paths. This paper introduces an approach for transforming a class of SDL specification in order to eliminate or reduce the infeasible path problem. This approach is divided into two phases in order to aid generality. First the SDL specification is rewritten to create a normal form extended finite state machine (NF-EFSM). This NF-EFSM is then expanded in order to produce a state machine in which the test criterion may be satisfied using paths that are known to be feasible. The expansion process is guaranteed to terminate. Where the expansion process may lead to an excessively large state machine, this process may be terminated early and feasible paths added. The approach is illustrated through being applied to the *Initiator* process of the *Inres* protocol.

*Key words:* SDL, test generation, extended finite state machine, infeasible path problem<sup>1</sup>.

---

## 1 Introduction

Testing is a vital part of the software development process. The test process typically is, however, time consuming, error-prone, and expensive. While these problems may be overcome, or reduced, by introducing test automation, test

---

<sup>1</sup> This is a preliminary version of the following paper: R.M. Hierons, T.-H. Kim, and H. Ural, 2004, On The Testability of SDL Specifications, *Computer Networks*, 44 5, pp. 681-700.

automation must be based on some source of information. One such source of information is a formal or semi-formal specification.

Many systems have some internal state that affects and is affected by the system's operations. Such state-based systems are often specified using an extended finite state machine (EFSM) based language such as SDL [11]. An SDL specification may be rewritten to form an EFSM which may act as the basis for automating or semi-automating testing [2,14].

When testing from an EFSM based language it is usual to generate a set of paths through the EFSM. Test data is then produced to trigger these paths. Each path contains a sequence of transitions, each of which has a precondition or guard. As a consequence of this a path  $p$  defines a path condition  $c(p)$ : a condition that an input sequence  $x$  must satisfy in order for  $x$  to lead to  $p$  being followed. Thus, generating test data for a path  $p$  involves finding an input sequence that satisfies  $c(p)$ .

It is possible for a path  $p$  to be infeasible: no input sequence satisfies the condition  $c(p)$ . This is a consequence of the preconditions of more than one transition contributing to  $c(p)$ : some of these preconditions may conflict. While it might be reasonable to expect each transition in a specification to be feasible (i.e. can be executed under some condition that may occur) many specifications will contain infeasible paths. For example, a process that starts by trying to establish a connection may have some counter that starts at 0 and is incremented on each failed attempt. Suppose the process abandons this attempt to make a connection if the counter reaches some predefined value  $n$ . Then any path that requires  $m$  consecutive failed attempts, for  $m > n$ , is infeasible. The presence of infeasible paths may lead to there being no test data that triggers a path chosen in test generation. The problem of generating tests from an EFSM may thus be complicated by the presence of infeasible paths.

This paper introduces a new approach that expands an EFSM in order to bypass the infeasible path problem. The procedure is composed of two phases: building a normal form EFSM (NF-EFSM) from a specification and expanding the NF-EFSM to improve testability. The use of an NF-EFSM aids generality: once a specification has been transformed into this form the expansion procedure may be applied. Thus, in order to extend the results to some other specification language such as Z [18,19], VDM [12] or Statecharts [4], it is sufficient to find some mapping from specifications in that language to NF-EFSMs.

This paper extends the work of [6–8] on the refinement of an EFSM for the generation of executable tests. The work in this paper is most similar to that in [8]. There are two main differences. First, [8] does not give an algorithm for generating tests from a partially expanded EFSM, produced where it is not

feasible to fully expand the EFSM. Further, here alternative approaches are evaluated on the *Initiator* process of the *Inres* protocol.

This paper is organized as follows. Section 2 provides a brief overview of SDL and defines a normal form EFSM. Section 3 shows the generation of an NF-EFSM from an SDL specification. The expansion procedure, which forms the core of this paper, is proposed in Section 4. Section 5 compares the work in this paper to previous work on expanding EFSMs while Section 6 considers the problem of generating tests from an expanded EFSM or a partially expanded EFSM. Finally, in Section 7, conclusions are drawn.

## 2 SDL Specifications and Normal Form EFSMs

SDL is a specification and description language standardized by the ITU (International Telecommunication Union). An SDL specification is graphical and symbol-based but its data is described using abstract data types and ASN.1[10]. It can be seen as a set of EFSMs communicating with each other where each EFSM is described by its process diagram with several logical states and transitions between them.

Typically, the sequential behaviour of a formal specification can be represented as an EFSM. However, the operation within a transition of this EFSM may contain conditional statements that determine which behaviour, out of some set of behaviours, is applied. For example, the operation might contain an *if* statement. Such a transition may be replicated to give one transition for each behaviour, in order to ensure that each behaviour is tested. The expanded EFSM is an EFSM where transitions have no guard related to internal variables, in other words, they are always executable at the originating state. However, in order to obtain executable transitions, some states may have to be split and some transitions may have to be replicated. The process of expanding an EFSM to eliminate infeasible paths is the main topic of this paper. We will now define the notion of a Normal Form EFSM.

**Definition 1** *A normal form extended finite state machine (NF-EFSM)  $M$  is the 7-tuple  $(S, s_0, V, \sigma_0, P, I, O, T)$  where*

- $S$  is the finite set of logical states
- $s_0 \in S$  is the initial state
- $V$  is the finite set of internal variables
- $\sigma_0$  denotes the mapping from the variables in  $V$  to their initial values
- $P$  is the set of input and output parameters
- $I$  is the set of input declarations
- $O$  is the set of output declarations

- $T$  is the finite set of transitions.

The label of a transition  $t \in T$  is defined by the 5-tuple  $(s_s, g_I, g_D, op, s_f)$  in which:

- $s_s$  is the start state of  $t$ ;
- $g_I$  is the input guard which can be expressed as the 3-tuple  $(i, P^i, g_{P^i})$ , where
  - $i \in I \cup \{\text{NIL}\}$ ;
  - $P^i \subseteq P$ ; and
  - $g_{P^i}$  is the input parameter guard that is either nil or represented as a logical expression given in terms of variables in  $V'$  and  $P'$  where  $V' \subseteq V$ ,  $\emptyset \neq P' \subseteq P^i$ ;
- $g_D$  is the domain guard and is either nil or represented as a logical expression given in terms of variables in  $V''$  where  $\emptyset \neq V'' \subseteq V$ ;
- $op$  is the sequential operation which is composed of simple statements such as output statements and assignment statements; and
- $s_f$  is the final state of  $t$ .

Note that if  $P^i = \emptyset$ , then  $g_P = \text{NIL}$ . External events which may trigger state transitions of the system are represented as input declarations in an NF-EFSM (they are members of the set  $I$ ). Input parameters are the attributes or parameters of those external events.  $V$  contains all of the variables that occupy some memory in the system.

The label of a transition in an NF-EFSM has two guards that decide the feasibility of the transition: the input guard  $g_I$  and the domain guard  $g_D$ .  $g_I$  is the guard for inputs, or events, from the environment that must be satisfied in order for the transition to be executed. An input  $i$  is represented by ‘?’ which means ‘input  $i$  from the environment’. Some inputs may carry values of specific input parameters and  $g_I$  may guard those values with the input parameter guard  $g_P$ , such as  $p = 1$  where  $p \in P$ . The input guard  $(\text{NIL}, \emptyset, \text{NIL})$  represents no input being required, which makes the transition *spontaneous*.  $g_D$  is the guard, or precondition, on the values of variables in the system (e.g.  $v < 4$ , where  $v \in V$ ). Note that in order to satisfy the domain guard  $g_D$  of a transition  $t$  it may be necessary to take some specific path to the initial state of  $t$ .  $op$  is a set of sequential statements such as  $v := v + 1$  and  $!o$  where  $v \in V$ ,  $o \in O$ , and  $!o$  means ‘output  $o$  to the environment (at a specific output port)’.

Clearly, none of the spontaneous transitions in an NF-EFSM should be without any guards, i.e., uncontrollable. This observation leads to the following assumption.

**Assumption 1** An NF-EFSM does not have any transition with  $g_I = (\text{NIL}, \emptyset, \text{NIL})$  and  $g_D = \text{NIL}$ .

A transition in an NF-EFSM is *conditional* if its domain guard  $g_D$  is not NIL. A variable used in the domain guard of a transition in an NF-EFSM is called a *guard variable* of the transition. A variable in an NF-EFSM is a *control variable* if it is a guard variable of some transition of the NF-EFSM. A transition in an NF-EFSM is *unconditional* if its domain guard  $g_D$  is NIL.

The operation of a transition in an NF-EFSM has only simple statements such as output statements and assignment statements, that is, it has no branching statements such as ‘if ... else’, ‘case’, ‘for’, ‘repeat ... until’, and ‘do ... while’ statements. Therefore, an NF-EFSM has the following property.

**Property 1** When a transition in an NF-EFSM is executed, all the actions of the operation specified in its label are performed consecutively and only once.

**Definition 2** An NF-EFSM is deterministic if for every input sequence  $x$  there is no more than one output sequence that may be produced by the NF-EFSM in response to  $x$ .

**Definition 3** An NF-EFSM is strongly connected if for every ordered pair of states  $(s, s')$  there is some feasible path from  $s$  to  $s'$ .

**Assumption 2** An NF-EFSM is *deterministic* and *strongly connected*.

Loops may lead to the explosion of the state space of an EFSM and affect the executability of a transition. In the following, we analyze loops in an NF-EFSM and make a few assumptions on loops in order to simplify the problem studied in this paper. Future work will consider how these assumptions may be relaxed.

Let us start with the definition of several terms.

**Definition 4** In an NF-EFSM,

- The (global) control state of an NF-EFSM is a set  $G = V_C \cup \{\text{logical state variable}\}$  where  $V_C$  is the set of control variables and the logical state variable takes a value from  $S$ .
- a cycle is a path that starts and ends at the same state, i.e., its starting and terminating states are the same.
- a simple cycle or a loop is a cycle in which none of the states appears more than once, except the starting state which appears twice.
- a self loop is a loop that is constructed from one transition.
- a loop is unconditional if all of its transitions are unconditional; otherwise, it is conditional.

Note that by definition, any unconditional loop is an infinite loop: the number

of iterations is unbounded. There are two types of unconditional loop.

**Definition 5** *An unconditional loop is a Type 1 unconditional loop where each iteration of the loop generates the same global control state subspace.*

Such a loop does not cause the state space explosion of the NF-EFSM. We allow this type of unconditional loops in an NF-EFSM.

**Definition 6** *An unconditional loop is a Type 2 unconditional loop if some iteration of the loop generates a different global control state subspace.*

Type 2 unconditional loops will not be allowed in an NF-EFSM in order to avoid an infinite state space.

We differentiate between three types of conditional loops.

**Definition 7** *A conditional loop is a Type 1 conditional loop if the number of iterations of the loop is not bounded above and each iteration of the loop generates the same global control state subspace.*

**Definition 8** *A conditional loop is a Type 2 conditional loop if the number of iterations of the loop is not bounded above and some iteration of the loop generates a different global control state subspace.*

Type 1 and Type 2 conditional loops are thus equivalent to Type 1 and Type 2 unconditional loops and therefore we allow Type 1 conditional loops and we do not allow Type 2 conditional loops in an NF-EFSM. Two examples of Type 2 conditional loops are given in Figure 1 (here  $x$  and  $y$  are assumed to be control variables). The first example shows the case where the variables used in the domain guards are not modified. In the second case there are modifications to the variables used in the domain guards but here these modifications cannot contribute to the satisfaction of the condition to terminate the loop.

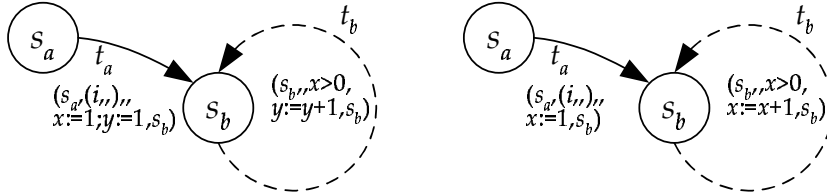


Fig. 1. Two examples of Type 2 conditional loops

**Definition 9** *A conditional loop is a Type 3 conditional loop if the number of iterations of the loop is bounded above.*

We identify two classes of Type 3 conditional loops: one class consists of single-transition loops, i.e., self-loops, and the other class consists of multiple-transition loops. We will only deal with single-transition loops; we leave the

handling of NF-EFSMs that have multiple-transition conditional loops with finite iteration for future research.

**Definition 10** A *Type 3 conditional loop* is said to be well-structured if it is a *self-loop*.

**Assumption 3** All loops within an NF-EFSM are either Type 1 unconditional loops, Type 1 conditional loops, or well-structured Type 3 conditional loops.

Note that a consequence of this assumption is that every path from the initial state back to the initial state, that is not a self-loop, returns the value of each state variable to its initial value.

### 3 Producing an NF-EFSM from an SDL Specification

A process diagram in an SDL specification is an EFSM. A transition from one logical state to another is described in a series of symbols, each representing an element of the transition. The guard of a transition is decided by both input symbols and decision symbols. In general, a transition has one input symbol, but may have several decision symbols. Moreover, there may be a cyclic path with a decision. To directly generate an NF-EFSM, the process diagram should be in the form of Figure 2(a). If an operation has complex elements such as multiple decision symbols, cyclic paths, timer operations, saves, and procedure calls it can be flattened using various techniques[20].

Domain propagation may also be used [1]. Domain propagation Partition individual operators such that their behaviour in a subdomain of the partition is uniform. Each such operator can be replaced by a disjunction of behaviours with preconditions. The following is an example.

$$y = |x|$$

$$\rightarrow ((x \geq 0) \wedge (y = x)) \vee ((x < 0) \wedge (y = -x))$$

Consider the process diagram specified in SDL of the *Initiator* process of the *Inres* protocol [9] shown in Figure 3. To build the NF-EFSM, timer operations are flattened as follows. For a timer  $T$ , we define a variable  $T$  that records the remaining time to the expiry of  $T$ . If there are more than two timers, we define another variable *min\_timer* that contains the minimum value of all currently active timeout periods [20]. The timer expiry input of  $T$  is changed to the input  $T\_expired$  and the statement ‘undef  $T$ ’. Undef applied to a variable  $x$  makes the value of  $x$  undefined. This is considered to be equivalent to assigning a special value to the variable  $x$ . The application of *Set* to a timer  $T$  is converted

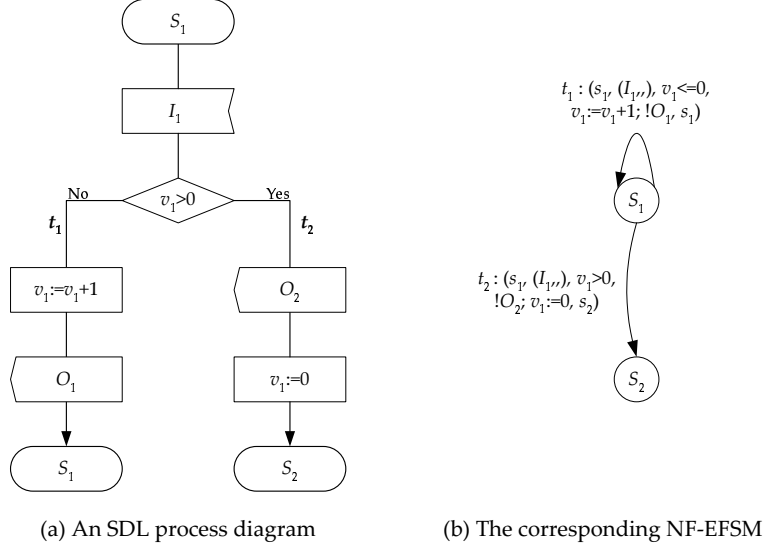


Fig. 2. An SDL process diagram and its representation in the NF-EFSM

to the assignment of the duration to variable  $T$ , and *reset* applied to timer  $T$  is converted into the statement ‘undef  $T$ ’. It is very difficult to flatten *save* operations in general. In this example, a *save* operation is used to keep the user data from being lost. In our example, that operation is removed in the NF-EFSM by assuming that the input queue from the user is controlled to send out ‘IDATreq’ signal only when *Initiator* is at the *connect* state. For testing a *save* operation of an input, feasible subpaths may be added to the NF-EFSM as new transitions each of which starts with some transition having the *save* operation and ends with some transition whose guard has the corresponding input.

The function ‘succ( $v$ )’ toggles between 0 and 1 for the value of a binary variable  $v$ . The task  $number := succ(number)$  is flattened as follows:

$$number = \begin{cases} 1 & \text{if } number = 0 \\ 0 & \text{if } number = 1 \end{cases}$$

The final NF-EFSM of *Initiator* process is shown in Figure 4.

#### 4 The Expansion Procedure

This paper focuses on the problem of producing an expanded EFSM given a specification of a deterministic system. The purpose of this expansion is to simplify test generation. In order to provide generality, the expansion is based on a two-phase transformation approach as shown in Figure 5. The

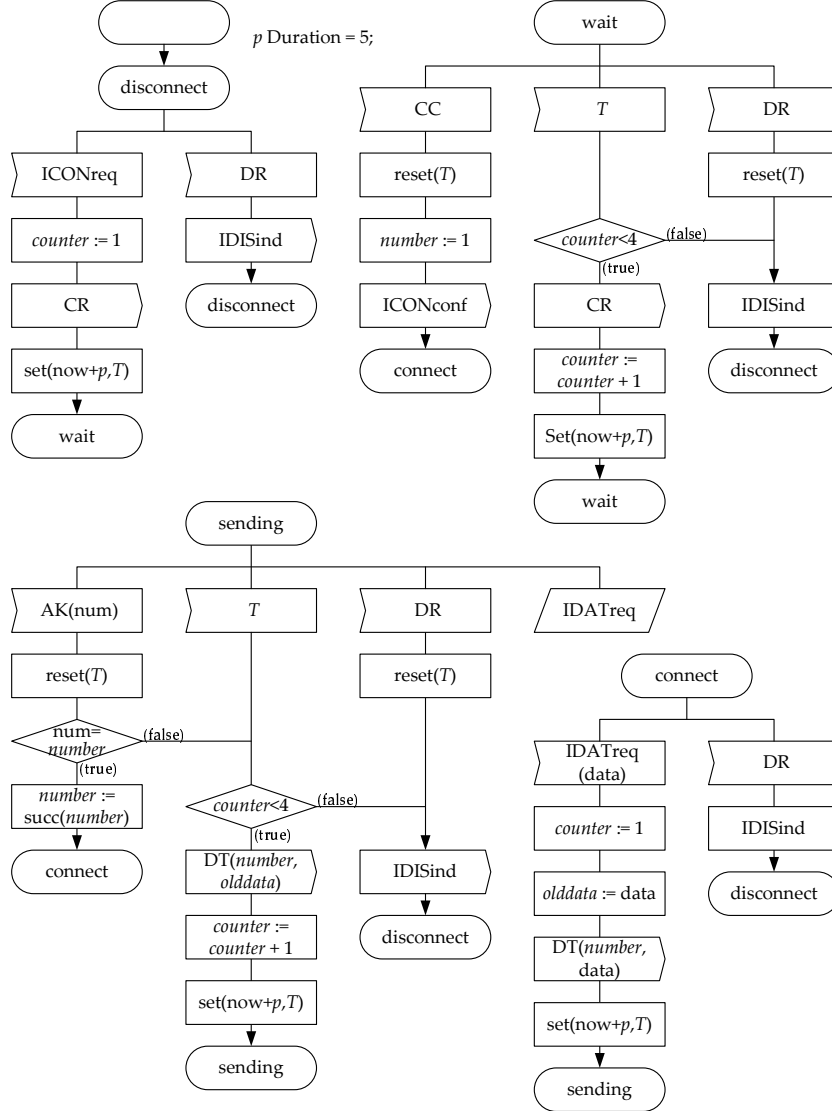


Fig. 3. The SDL diagram of *Initiator* process of *Inres* protocol

normalization phase of a specification varies according to its formal model but the expansion phase of an NF-EFSM is common for any specification. The motivation for using an NF-EFSM is as follows. First, the syntax of an NF-EFSM is independent of the syntax of the specification language used. Second, every operation of a transition in an NF-EFSM represents a *single* behaviour which can be executed if its guard is satisfied. In Section 6 it will be shown that this feature is important when applying data flow testing. Finally, most of the existing methods for test generation can be applied directly to an NF-EFSM even if we don't expand it.

This section describes the procedure that expands an NF-EFSM to form an Expanded EFSM (EEFSM) or a Partially Expanded EFSM (PEEFSM). The expansion procedure will be illustrated using the NF-EFSM in Figure 4 obtained from the SDL specification of the Initiator Process.

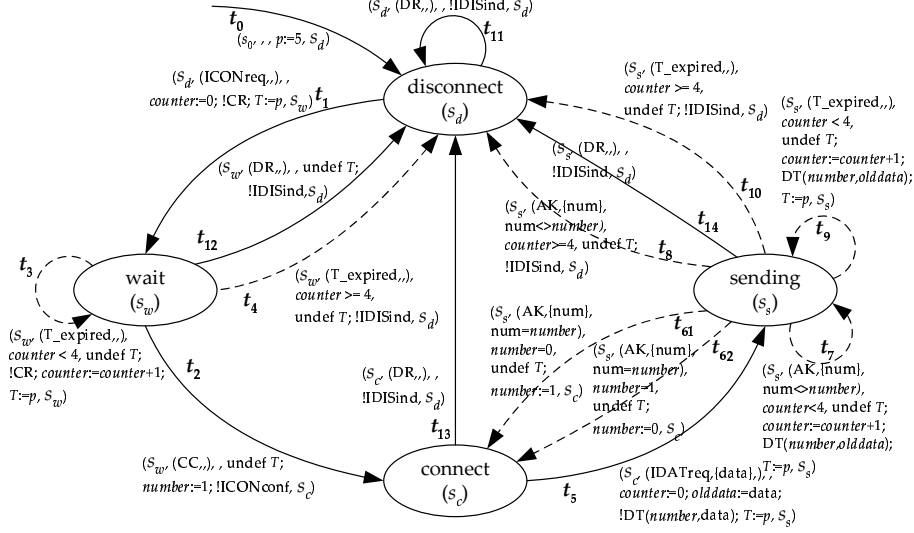


Fig. 4. The NF-EFSM of Initiator process of Inres protocol

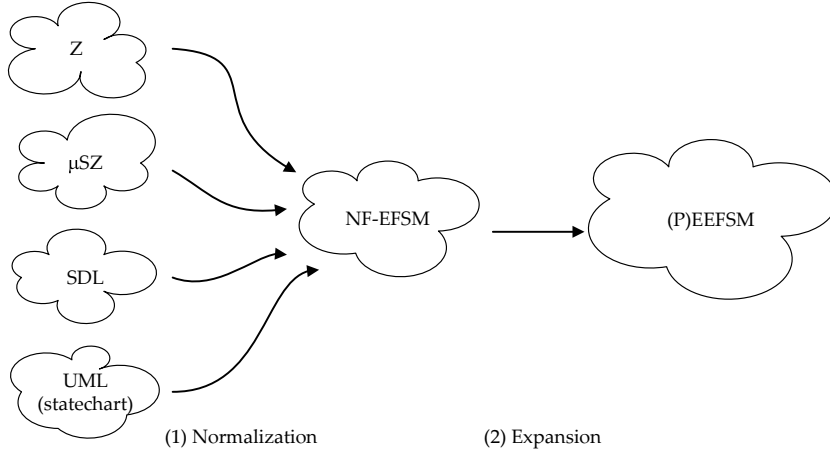


Fig. 5. The proposed approach: two-phase expansion

#### 4.1 Expansion of an NF-EFSM

##### 4.1.1 Notation

Before giving a detailed description of the expansion algorithm, we introduce some notation and functions. First,  $\mathcal{D}$  denotes the domain constructed from all the control variables in  $V$  and  $\Lambda$  denotes the domain constructed from all the input parameters in  $P$  that are used in the input guards. The subset of  $\mathcal{D}$  allowed at a state will be called the *domain* of the state.

Recall that the label of a transition  $t$  is  $(s_s, g_I, g_D, op, s_f)$  where  $s_s$  is the start state,  $g_I$  is the input guard,  $g_D$  is the domain guard,  $op$  is the sequential operation, and  $s_f$  is the final state of  $t$ . In this paper, we use the term *precondition* of a transition  $t_i$ , denoted  $P_i$ , to mean the domain guard  $g_D$  of  $t_i$ . The term

*parameter condition* of a transition  $t_i$ , denoted by  $\lambda_i$ , is the input parameter guard  $g_P$  of  $t_i$ .

The unary *dom* operator takes a logical expression and returns the subdomain of  $\mathcal{D}$  that satisfies this condition while the unary *cond* operator takes a subdomain of  $\mathcal{D}$  and returns the corresponding logical expression. The *postcondition* of a transition  $t_i$ , denoted by  $Q_i(\cdot) : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\Lambda) \rightarrow \mathcal{P}(\mathcal{D})$ , is the function that derives a domain in  $\mathcal{D}$ , according to the operation  $op_i$  of the transition  $t_i$ , given two subdomains of  $\mathcal{D}$  and  $\Lambda$  respectively.  $d(\cdot) : S \rightarrow \mathcal{P}(\mathcal{D})$  is the domain function of a state, and  $s_{ST}(\cdot) : T \rightarrow S$  and  $s_{FN}(\cdot) : T \rightarrow S$  are the starting state and final state functions of a transition, respectively.

The algorithm requires that all the postcondition functions and their inverse functions can be evaluated symbolically in any domain considered.

#### 4.1.2 Algorithm

**Step.1** Partition the domain of a state  $s$  that has at least two conditional transitions originating from it as follows: Let the conditional transitions  $t_1, t_2, \dots, t_n (n \geq 2)$  originating from state  $s$  have preconditions  $P_1, P_2, \dots, P_n$  respectively.

Each subdomain,  $\{\mathcal{P}_X^s | X \subseteq \{1, \dots, n\} \wedge X \neq \{\}\}$  is given by

$$\mathcal{P}_X^s = \text{dom}((\bigwedge_{i \in X} P_i) \wedge (\bigwedge_{i \notin X} \neg P_i)).$$

For example, if an operation at a state  $s$  is rewritten as  $\bigvee_{1 \leq i \leq 3} (P_i \wedge Q_i)$ , a partition of the domain of state  $s$  by the operation is

$$\{\mathcal{P}_{\{1\}}^s, \mathcal{P}_{\{2\}}^s, \mathcal{P}_{\{3\}}^s, \mathcal{P}_{\{1,2\}}^s, \mathcal{P}_{\{2,3\}}^s, \mathcal{P}_{\{1,3\}}^s, \mathcal{P}_{\{1,2,3\}}^s\}.$$

Each  $\mathcal{P}_X^s$  in the domain of state  $s$  can be depicted as Figure 6.

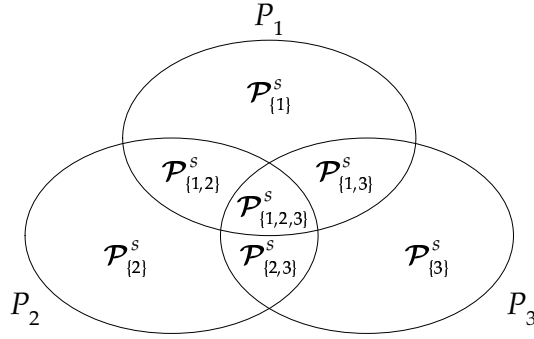


Fig. 6. An example of partitioning

The number of disjoint subdomains is at most  $2^n - 1$  but may be fewer because some of them may be empty.

Then, if the final non-empty disjoint subdomains are  $\mathcal{P}_1^s, \dots, \mathcal{P}_m^s (m \leq 2^n - 1)$ , split the state  $s$  to  $s_1, \dots, s_m$  whose domains are  $\mathcal{P}_1^s, \dots, \mathcal{P}_m^s$ , respectively.

If this is the first iteration, repeat this step for all the states from

which there are outgoing conditional transitions. After the first iteration, priority is given to states that are not within any well-structured loop, if there exist such states; otherwise, the state to be split is selected among states that are within well-structured loops.

**Step.2** Rearrange transitions related to the split states. If a state  $s_i$  is split into  $n(\geq 2)$  states,  $s_{i_1}, \dots, s_{i_n}$ , remove each transition  $t_j$  going from or to the state  $s_i$ . Then, for each removed transition  $t_j$  going from the state  $s_i$  to a state  $s_f (\neq s_i)$ , make  $n$  temporary transitions going from  $s_{i_k}$  ( $1 \leq k \leq n$ ) to  $s_f$  whose labels are the same as that of the removed transition. For each removed transition  $t_j$  going to the state  $s_i$  from a state  $s_s (\neq s_i)$ , make  $n$  temporary transitions going from  $s_s$  to  $s_{i_k}$  ( $1 \leq k \leq n$ ) whose labels are the same as that of the removed transition. For each removed transition  $t_j$  going from and to the same state  $s_i$ , a self loop, make  $n^2$  temporary transitions going from each  $s_{i_k}$  ( $1 \leq k \leq n$ ) to each  $s_{i_{k'}}$  ( $1 \leq k' \leq n$ ) whose labels are the same as that of the removed transition.

**Step.3** For each temporary transition  $t_i$ , there are only two conditions on the relationship between  $d(s_{ST}(t_i))$  and  $dom P_i$  since  $s_{ST}(t_i)$  is defined by a subdomain  $\mathcal{P}_X^{s_{ST}(t_i)}$  for some  $X$ :  $d(s_{ST}(t_i)) \subseteq dom P_i$  or  $d(s_{ST}(t_i)) \cap dom P_i = \emptyset$ .

Therefore, for each temporary transition  $t_i$ , make it permanent or discard it depending on the following cases:

- Case A. If  $dom P_i \cap d(s_{ST}(t_i)) = \emptyset$  or  $Q_i(d(s_{ST}(t_i)), dom \lambda_i) \cap d(s_{FN}(t_i)) = \emptyset$ , discard  $t_i$ .
- Case B. If  $dom P_i \supseteq d(s_{ST}(t_i))$  and  $Q_i(d(s_{ST}(t_i)), dom \lambda_i) \subseteq d(s_{FN}(t_i))$ , make  $t_i$  unconditional.
- Case C. If  $dom P_i \supseteq d(s_{ST}(t_i))$  and  $Q_i(d(s_{ST}(t_i)), dom \lambda_i) \not\subseteq d(s_{FN}(t_i))$  and  $Q_i(d(s_{ST}(t_i)), dom \lambda_i) \cap d(s_{FN}(t_i)) \neq \emptyset$ ,
  - If  $dom P'_i \supseteq d(s_{ST}(t_i))$  then make  $t_i$  unconditional.
  - If  $dom P'_i \not\supseteq d(s_{ST}(t_i))$  then make  $t_i$  conditional with domain guard  $P'_i$ .

Here  $P'_i = d(s_{ST}(t_i)) \cap Q_i^{-1}(d(s_{FN}(t_i)))$ .

Note that  $P_i$  and  $\lambda_i$  are the precondition and the parameter condition of  $t_i$  respectively, and  $Q_i(\cdot)$  is the postcondition of transition  $t_i$ .

**Step.4** If the initial state is split, determine which of the split states is now the initial state. Remove all states that cannot be reached from the initial state. Then, if Condition A or Condition B is satisfied, terminate. Otherwise, return to Step 1.

- (Condition A. Complete termination) There are no conditional transitions.
- (Condition B. Reasonable termination) All the remaining conditional transitions in the present PEEFSM are conditional transitions constructing well-structured loops in the original NF-EFSM, and further expansion is considered, by the user, to be impractical or unnecessary

because sufficient unconditional transitions are obtained to satisfy the selected test coverage criterion.

The following property is an immediate consequence of the restrictions placed on loops.

**Property 2** Every NF-EFSM may be expanded to form a finite EEFSM.

#### 4.1.3 Justification

The algorithm attempts to construct an EEFSM from a given NF-EFSM by partitioning the domain of each state with the preconditions of its conditional transitions. When a state  $s$  is split in order to change outgoing conditional transitions into unconditional ones, several conditional transitions may be generated from the transitions that end at  $s$ . So, the algorithm may have to split some states repeatedly until one of the termination conditions of Step.4 is satisfied. The repetitive splitting of states for building the EEFSM may yield a large EEFSM. In some cases it may not be practical to produce the complete EEFSM and here the reasonable termination condition (Condition B) allows the tester to terminate the expansion.

Figure 7 shows an example of the reasonable termination with a well-structured loop. In this figure and the following figures,  $g_I = (\text{NIL}, \emptyset, \text{NIL})$ ,  $g_D = \text{NIL}$ , and  $g_P = \text{NIL}$  are represented by blanks and NIL or empty components are also represented by blanks. In addition, dotted arrows are used to indicate that the transition is conditional. From state  $s_b$ , two conditional transitions  $t_b$  (a self loop  $\ell$ ), and  $t_c$  originate. In the first iteration of the algorithm,  $s_b$  is split to  $s_{b_1}$  and  $s_{b_2}$  according to the preconditions of  $t_b$  and  $t_c$ . Then, the transition  $t_b$  is replicated to four temporary transitions whose starting and terminating states are  $s_{b_1}$  and  $s_{b_1}$ ,  $s_{b_1}$  and  $s_{b_2}$ ,  $s_{b_2}$  and  $s_{b_1}$ , and  $s_{b_2}$  and  $s_{b_2}$ , respectively. Among those temporary transitions,  $t_{b_1}$  and  $t_{b_2}$  become conditional. After the first iteration of the algorithm, there is a well-structured loop  $\ell'$  instead of  $\ell$ . The well-structured loop will be modified repeatedly until it is eventually changed to a transition at the  $n$ -th iteration. However, if  $n$  is large, it may be appropriate to use the reasonable termination condition. The PEEFSM generated by the first iteration satisfies the reasonable termination condition and the decision to terminate is made by the user or some heuristic. Section 6 will consider the problem of generating tests from a PEEFSM.

#### 4.2 An Example

Consider the NF-EFSM given in Figure 4. The application of the expansion algorithm to this NF-EFSM progresses as follows:

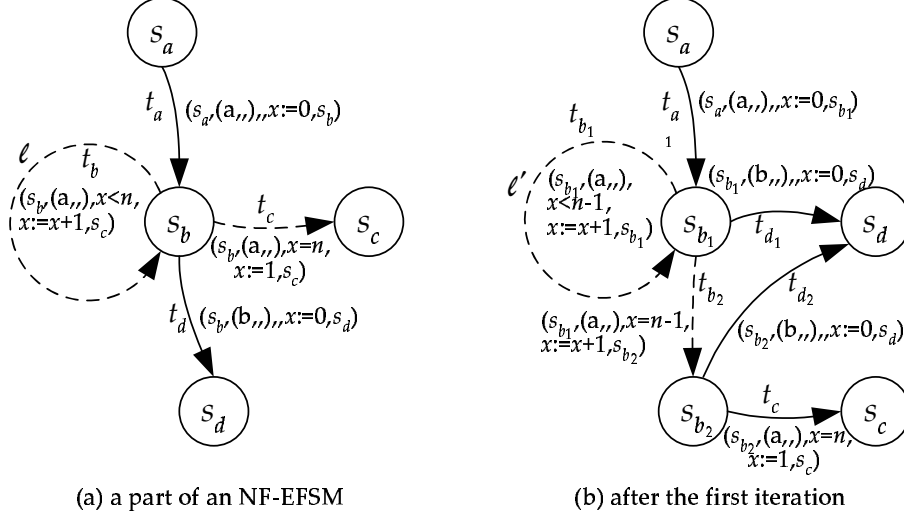


Fig. 7. An example for Condition B

At Step.1, the domain of state *wait* is partitioned according to the disjoint preconditions of two conditional transitions,  $P_1 = (\text{counter} < 4)$  and  $P_2 = (\text{counter} \geq 4)$ . So *wait* is split as follows.

$$\text{wait}_1 : (\text{counter} < 4)$$

$$\text{wait}_2 : (\text{counter} \geq 4)$$

Since this is the first iteration, the domain of state *sending* is also partitioned according to the preconditions,  $P_1 = (\text{counter} < 4)$ ,  $P_2 = (\text{counter} \geq 4)$ ,  $P_3 = (\text{number} = 0)$ , and  $P_4 = (\text{number} = 1)$  from conditional transitions  $t_{61}, t_{62}, t_7, t_8, t_9$ , and  $t_{10}$ , as follows. Among  $15 (= 2^4 - 1)$  candidate subdomains, there are four non-empty subdomains and thus the state *sending* is split to form the following four states.

$$\text{sending}_1 : (\text{counter} < 4 \wedge \text{number} = 0)$$

$$\text{sending}_2 : (\text{counter} < 4 \wedge \text{number} = 1)$$

$$\text{sending}_3 : (\text{counter} \geq 4 \wedge \text{number} = 0)$$

$$\text{sending}_4 : (\text{counter} \geq 4 \wedge \text{number} = 1)$$

At Step.3, 18 temporary transitions become unconditional ones (Case B), 12 become conditional ones (Case C), and the other ones are discarded (Case A). At the end of this step, a PEEFSM of *Initiator* process is generated as shown in Figure 8, where the labels of transitions are not shown in order to aid simplicity. In this example, the copies of a transition are distinguished by the use of labels.

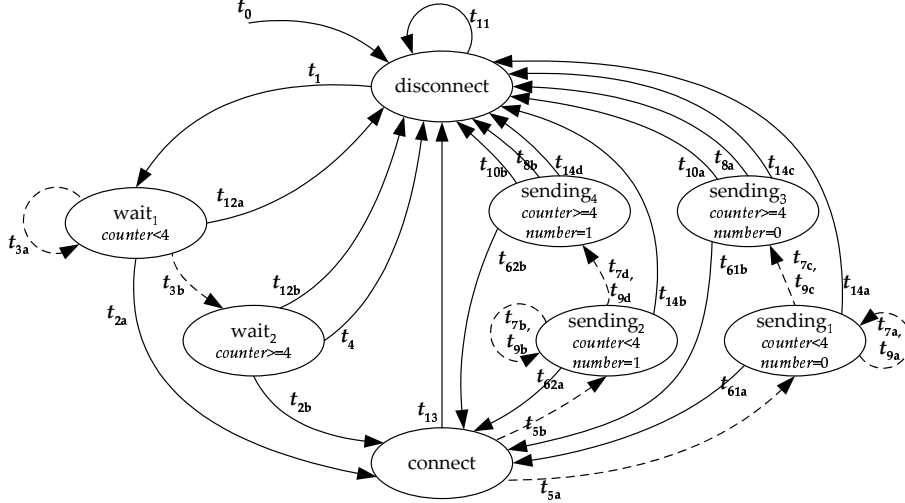


Fig. 8. On expanding process of *Initiator* process : after the first iteration

At Step.4, Conditions A and B are not satisfied because there are conditional transitions,  $t_{5a}$  and  $t_{5b}$  which do not originate from the starting state of any well-structured loop. We still have 12 conditional transitions and we start the second iteration of the algorithm.

In the second iteration of the algorithm, at Step.1, *connect* is selected and this is partitioned according to the preconditions  $P_1 = (number = 0)$  and  $P_2 = (number = 1)$ . This gives the following two states.

$connect_1 : (number = 1)$

$connect_2 : (number = 0)$

At Step.3, 10 temporary transitions become unconditional (Case B) and the others are discarded (Case A). As a consequence of state splitting the transitions  $t_{5a}$  and  $t_{5b}$  became unconditional. At the end of this step, a PEEFSM of the *Initiator* process is generated as shown in Figure 9.

At Step.4, Condition A is not satisfied but Condition B may be satisfied because  $t_{5a}$  and  $t_{5b}$  have been changed to unconditional transitions. While this process may continue to produce an EEFSM, it will terminate here in order to illustrate issues regarding generating tests from PEEFSMs.

## 5 Comparisons with Previous Work

Two papers [6,7] presented test generation methods from a Z specification and a  $\mu$ SZ specification, respectively. They partitioned the domain of the input or the internal memory according to the preconditions of the operations.

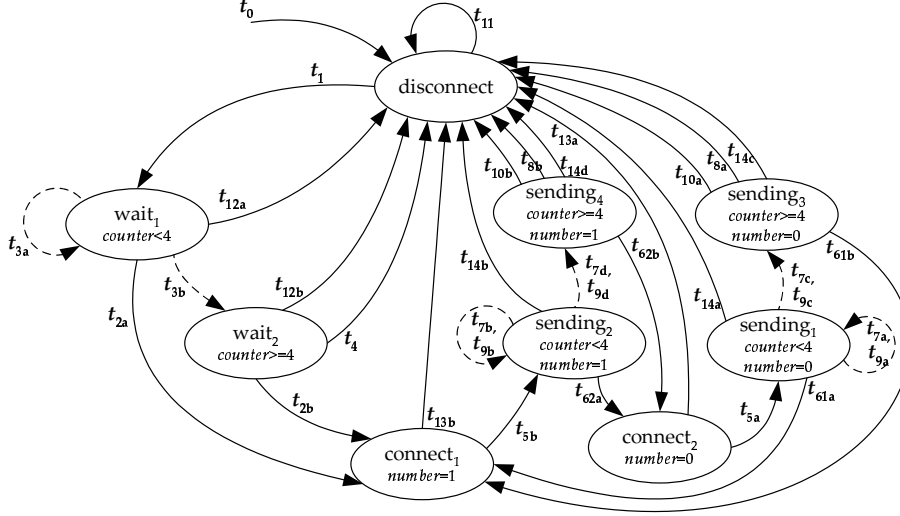


Fig. 9. On expanding process of *Initiator* process : after the second iteration

Test cases, for control flow testing, are generated from the resultant EFSM. However, these approaches are specific to the specification language used.

Hierons et al. [7] refine the EFSM by data abstraction, which is similar to the first iteration of our algorithm. It does not go further because the repetitive refining may not terminate. However, as discussed in Section 4.1.3, the algorithm in this paper is guaranteed to terminate under the assumptions made and it may be terminated where further expansion may make the number of states excessive. The approach of Hierons et al. [7] may also introduce nondeterminism which is not inherent in the system. This complicates test generation.

Recent work by Uyar and Duale has considered the problem of eliminating the infeasible path problem for EFSMs where it is known that all operation and guards are linear [21]. The assumptions made in this paper are quite different: rather than assume linearity, restrictions are placed on the structure of the EFSM.

Henniger transforms an Estelle specification to form an equivalent EEFSM [5]. The transformation is feasible under the assumption that the control variables have finite domains. However, it first generates a very large FSM: for every state in the EFSM and every combination of values for the control variables, it produces a state in the EEFSM. This EEFSM is then minimized. Naturally, this approach may suffer from the state space explosion problem.

## 6 Test Generation

This section will consider the problem of generating a test to satisfy the all-uses criterion. The all-uses criterion considers definitions and uses of variables. An assignment  $x := e$  is a definition of  $x$  and a use of each variable referenced by  $e$ . Outputs and guards are uses of the variables referenced. A path is definition clear with respect to a variable  $x$  if it contains no definitions of  $x$  after its first transition. Then a feasible du-pair consists of an ordered pair  $(t_1, t_2)$  of transitions where there is some variable  $x$  that is defined at  $t_1$  and used at  $t_2$  such that there is a feasible definition clear path with respect to  $x$  that starts with  $t_1$  and ends with  $t_2$ . The all-uses coverage is the proportion of feasible du-pairs executed in testing. The all-uses criterion is satisfied if and only if all feasible du-pairs are covered in testing.

Due to the feasibility of all paths in an EEFSM, test generation from an EEFSM is straightforward. If we have a PEEFSM which is constructed by using the reasonable termination condition, there are some conditional transitions. In this case, it is possible to use one of the following solutions:

- (1) Try to generate test paths which do not traverse conditional transitions.
- (2) Resume the expansion algorithm to get the complete but potentially very large EEFSM.
- (3) If test generation is based on a given test coverage criterion such as all-uses, a PEEFSM may be transformed to an EEFSM that is smaller than the complete EEFSM. Such an EEFSM is not equivalent to the original NF-EFSM, but it has a set of unconditional paths starting at the initial state that, between them, satisfy the test coverage criterion.

The first approach may be a reasonable and practical solution. However, this may lead to a low test coverage. We will now compare these approaches for the *Initiator* process of the *Inres* protocol. Table 1 contains a summary of the results of the comparison, which will now be described in more detail.

Table 1

The number of du-pairs determined to be executable

| The number of                          | NF-EFSM  | PEEFSM<br>(after 2nd iteration) | Complete<br>EEFSM | Transformed<br>EEFSM |
|--|----------|---------------------------------|-------------------|----------------------|
| du-pairs                               | 61       | 164                             | 389               | 196                  |
| executable du-pairs<br>(unconditional) | 6 (9.8%) | 21 (12.8%)                      | 389 (100%)        | 196 (100%)           |
| executable du-pairs<br>(NF-EFSM basis) | 6 (9.8%) | 13 (21.3%)                      | 61 (100%)         | 61 (100%)            |

The above result was obtained from the *Initiator* process of the *Inres* protocol shown in Figure 4. After two iteration of the expansion algorithm, Condition B was satisfied and the PEEFSM shown in Figure 9 was generated. We used this PEEFSM as a target for comparison. The result shows the number of the required du-pairs that are determined to be executable, where a du-pair is said to be *executable* if there is a triple (unconditional preamble path starting from the initial state, unconditional def-clear path, unconditional postamble path going to the terminating state) for the du-pair. These three paths may be combined to form an unconditional path that covers the du-pair and returns to the initial state. As shown in Figure 4, 8 of the 16 transitions in the NF-EFSM of the *Initiator* process are conditional. We can easily determine that 6 out of 61 du-pairs are executable.

The PEEFSM generated after the second iteration has 164 du-pairs, and only 21 du-pairs are determined to be executable among those. The expansion has lead to seven additional du-pairs being determined to be executable. After the fifth iteration, we have the complete EEFSM which has 389 executable du-pairs. It should be noted that 164 du-pairs in the PEEFSM and 389 du-pairs in the EEFSM contain multiple copies of the 61 du-pairs in the NF-EFSM. It is sufficient to have one executable copy of each du-pair.

The second solution, which corresponds to the third column in Table 1, involves producing an EEFSM. In some cases this may be considered to be justified by the test requirements or risk.

The last solution is not a general solution: a transformation method that targets the test criterion is applied. Essentially, the PEEFSM is transformed by adding unconditional paths with the intention of making the test criterion satisfiable with unconditional paths. A transformation method is proposed in this section. It generates a much smaller EEFSM, which has only 196 executable du-pairs, than the complete EEFSM. The transformed EEFSM allows the all-uses criterion to be satisfied using unconditional paths (see the 4th column of Table 1).

In the following two subsections, we discuss test coverage and test generation. Since test generation for control flow test was discussed in [7], here we focus on data flow test generation. Section 6.2 gives a method for generating tests from a PEEFSM produced by the reasonable termination condition.

### 6.1 Test coverage

Where a function is used within the definition of an operation, and this has a number of separate behaviours, the transition need not be split when generating the NF-EFSM. However, domain propagation may be applied to split the

transition in order to increase test coverage. Consider the ‘succ( $\cdot$ )’ function in the *Initiator* process. This can be written by using conditional statements or by using the modular function. In the first case, domain propagation should be applied, while in the second case, domain propagation is not necessary if the modular function is a built-in function of the description tool for SDL. In the example, we assumed that the function was written in the first form.

It is worth noting that when rewriting an EFSM to an NF-EFSM, a transition is split where it has a number of separate behaviours with conditions. This assists data flow testing when the transition’s operation has a number of sub-behaviours which differ in either the variables referenced or the variables defined. For example, suppose a transition  $t$  whose domain guard is  $g_D$ , has the operation defined by `if (x>0) y:=a; else z:=b;`. In forming an NF-EFSM,  $t$  is split into two transitions:  $t_1$  whose domain guard is  $x > 0 \wedge g_D$  and operation is `y:=a;` and  $t_2$  whose domain guard is  $x \leq 0 \wedge g_D$  and operation is `z:=b;`. This eliminates a potential problem in data flow testing: if  $t$  is not split, the data dependencies exercised by executing  $t$  within a test sequence may depend upon the value of  $x$  when  $t$  is executed. It may thus appear that a data dependence has been exercised, due to a path being traversed, when this data dependence has not been covered.

## 6.2 Test generation for data flow test

When we generate a manageable size EEFSM, test generation for data flow testing is straightforward because all paths are executable in that EEFSM. We simply find a set of paths satisfying the required test selection criterion. If instead we have a PEEFSM we may be able to transform that PEEFSM into a transformed EEFSM according to a specific test criterion and generate test cases satisfying that criterion. Naturally, the transformation applied will depend upon properties of the specification: we cannot expect there to be an algorithm that achieves this for all specifications. This section gives a transformation that, under the conditions outlined earlier and the one given below, allows the all-uses criterion to be satisfied. Future work will consider alternative conditions and corresponding transformation algorithms.

**Assumption 4** Suppose that well-structured Type 3 loop  $\ell$  starts and ends at state  $s$ . Then every path, from the initial state to  $s$ , initialises each state variable  $v$ , mentioned by the guard of  $\ell$ , to a constant.

The above guarantees that the value for the state variables in  $V$  mentioned in the guard of  $\ell$  is defined by the path taken. However, different paths may lead to different values for these variables.

The transformation algorithm is iterative. Each iteration involves choosing some state  $s_1$  with one or more conditional transitions leaving it and, on the basis of this, transforming the PEEFSM by splitting  $s_1$  and replacing the conditional transitions with paths generated by the unfolding of the conditional loops at  $s_1$ . This is outlined in Figure 10.

We will now explain how the state, considered in the current iteration, is chosen. When a state  $s$  is split in forming the PEEFSM, a set  $\{s_1, \dots, s_n\}$  of states is formed. Possibly, some states are unreachable and so are deleted. At least one of the states in  $\{s_1, \dots, s_n\}$  will be reached by no conditional transitions from other elements of  $\{s_1, \dots, s_n\}$ . Such a state  $s_1$  is called a *head state*. It is straightforward to see that there is some state  $s$  of the original EFSM with corresponding head state  $s_1$  in the PEEFSM that is reachable, from the initial state, using one or more unconditional paths. Such a state may be found through a breadth-first search starting at the initial state  $s_0$ . Such a head state  $s_1$ , and the corresponding set  $\{s_1, \dots, s_n\}$ , is then considered.

At  $s_1$ , there may be  $m$  well-structured loops which will be called  $\ell_1, \dots, \ell_m$ . Further, at state  $s_1$ , there are  $m$  conditional transitions going to  $s_2, \dots, s_{n'}$  ( $n' \leq n$ ) that are copies of the  $m$  well-structured loops and we call these  $t_1, \dots, t_m$ . In addition, there may be unconditional transitions  $t_1^o, \dots, t_w^o$  originating from  $s_1$  and unconditional transitions (or paths)  $T_1, \dots, T_u$  terminating at  $s_1$ . Those transitions are drawn shaded because their transformation is not shown in Figure 10(b).

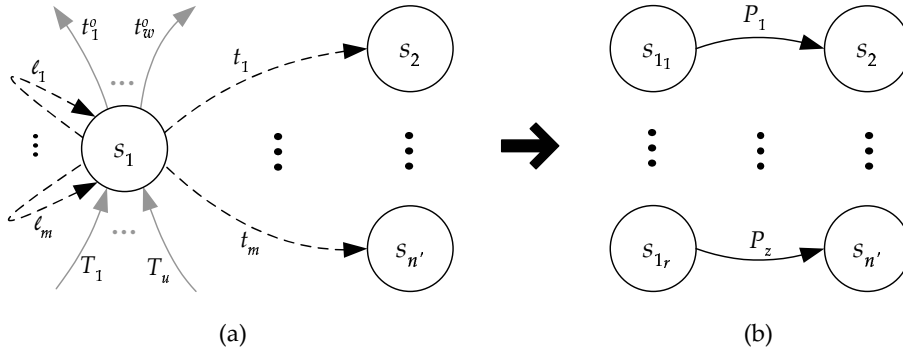


Fig. 10. Transformation of conditional transitions in a PEEFSM

The transformation algorithm replaces  $\ell_1, \dots, \ell_m$  and  $t_1, \dots, t_m$  by several unconditional paths. The paths added are designed to allow the all-uses criterion to be satisfied using these paths in place of the conditional transitions between the states in  $\{s_1, \dots, s_n\}$ .

Since  $s_1$  will be reached using paths from  $\{T_1, \dots, T_u\}$ , if the postconditions of  $T_1, \dots, T_u$  are not the same, the transformation algorithm splits  $s_1$  into  $s_{11}, \dots, s_{1r}$  according to the postconditions of  $T_1, \dots, T_u$ . Unconditional paths  $P_1, \dots, P_z$  ( $z \geq m$ ) are then constructed; these replace  $\ell_1, \dots, \ell_m$  and  $t_1, \dots, t_m$  as shown in Figure 10(b). Here assumption 4 is important since it guarantees

that the values of the state variables, mentioned in the guards of the loop, are fully defined by a path  $T_i$  and thus the process of unfolding the loops generates unconditional paths.

The transformation procedure proceeds as follows. First, the chosen head state  $s_1$  is split according to the postconditions of some unconditional paths going to  $s_1$  (Step.1). The PEEFSM may have to be rearranged again due to the states split at Step.1 (Step.2). Then, the procedure adds a set of unconditional paths that cover all du-pairs that are not covered by the paths of unconditional transitions in the PEEFSM (Step.3 and 4). The final transformed EEFSM is completed by removing isolated states and transitions that enter and leave isolated states.

**Step.1** Let  $s_1$  be a head state that has one or more conditional transitions leaving it and that is reachable, by unconditional paths, from the initial state. Let  $T_1, \dots, T_u$  denote unconditional paths that reach  $s_1$ . Let  $T_1, \dots, T_u$  have distinct postconditions  $Q^1, \dots, Q^u$  with respect to the set  $V'$  of control variables mentioned in the guards of the self-loops from  $s_1$ . Then, partition the domain of  $s_1$  into  $dom(Q^1), \dots, dom(Q^u)$ .

**Step.2** Execute the algorithm in Section 4.2.2 from Step.2 with the reasonable termination condition.

**Step.3** Let  $\ell_i (1 \leq i \leq m')$  denote a well-structured loop in the present PEEFSM which starts at a state  $s_{1j} (1 \leq j \leq r)$ . For every unconditional transition  $t_k^o (1 \leq k \leq w)$  originating from  $s_{1j}$ , add a path going from  $s_{1j}$  to  $s_{FN}(t_k^o)$  by concatenating  $\ell_i$  and  $t_k^o$  if there is a path starting with  $t_k^o$  within which a variable defined in  $\ell_i$  is used without being re-initialized previously.

**Step.4** Let  $S_1 = \{s_{11}, \dots, s_{1r}\}$ , and  $S_2 = \{s_2, \dots, s_n\}$ . Let  $\ell_i (1 \leq i \leq m')$  denote a well-structured loop in the present PEEFSM which starts from a state  $s_{1j} \in S_1$  and whose replicated conditional transition  $t_i$  terminates at  $s_l \in S_2$ . Let  $\ell_{i_1}$  and  $\ell_{i_2} (1 \leq i_1, i_2 \leq m')$  denote two well-structured loops in the present PEEFSM which start from  $s_{1j}$  and whose replicated conditional transitions  $t_{i_1}$  and  $t_{i_2}$  terminate at  $s_{l_1}, s_{l_2} \in S_1$  respectively. Then, construct a minimal number of paths composed of possible combination of the loop(s) to satisfy the following requirements.

- (1) There must be at least one path that starts with  $\ell_i$  and goes from  $s_{1j}$  to a state in  $S_2$ .
- (2) There must be at least one path that ends with a copy of  $\ell_i$  (possibly  $t_i$ ) and goes from a state in  $S_1$  to  $s_l$ .
- (3) There must be at least one path that contains a subpath composed of the concatenation  $(\ell_i, \ell_i)$  and goes from a state in  $S_1$  to a state in  $S_2$ .
- (4) There must be at least one path that contains a subpath composed of a concatenation  $(\ell_{i_1}, \ell_{i_2})$  that goes from a state in  $S_1$  to a state

in  $S_2$ , if, for an unconditional path  $T_k(1 \leq k \leq u)$  used at Step.1, the postcondition of  $T_k$  followed by  $\ell_{i_1}$  implies  $P_{i_2}^\ell \vee P_{i_2}^t$ , where  $P_{i_2}^\ell$  and  $P_{i_2}^t$  are the preconditions of  $\ell_{i_2}$  and  $t_{i_2}$  respectively.

- (5) There must be at least one path that contains a subpath composed of a concatenation  $(\ell_{i_2}, \ell_{i_1})$  and goes from a state in  $S_1$  to a state in  $S_2$ , if, for an unconditional path  $T_k(1 \leq k \leq u)$  used at Step.1, the postcondition of  $T_k$  followed by  $\ell_{i_2}$  implies  $P_{i_1}^\ell \vee P_{i_1}^t$ , where  $P_{i_1}^\ell$  and  $P_{i_1}^t$  are the preconditions of  $\ell_{i_1}$  and  $t_{i_1}$ .

Note that paths found to satisfy points 3-5 may also cover a number of the requirements in points 1 and 2. Observe that due to Assumption 4, these additional paths are unconditional.

**Step.5** Remove all conditional transition between the states in  $S_1 \cup S_2$ .

**Step.6** Remove the states that cannot be reached from any other states or do not have any outgoing transitions. Then, remove all the transitions whose originating or terminating states do not exist. If there are no well-structured loops, terminate; otherwise go to Step.1.

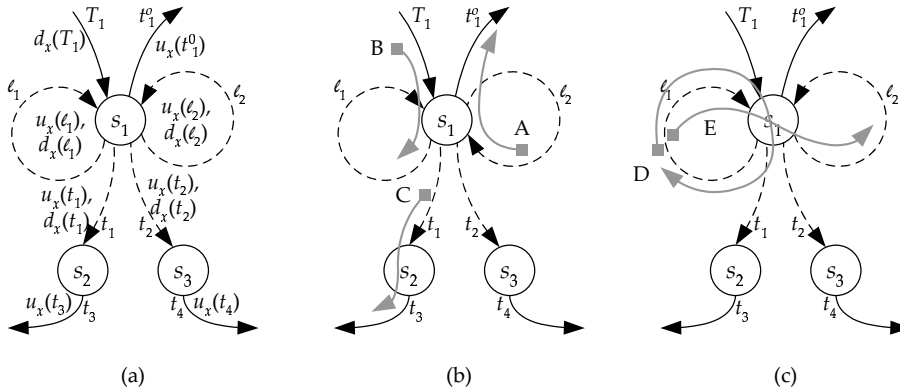


Fig. 11. Types of du-pairs to be considered

Steps 3 and 4 of the algorithm show the kinds of du-pairs we should consider when constructing unconditional paths to cover the required du-pairs. Since a well-structured loop has a ‘use’ and a ‘def’ of its guard variable, all distinct du-pairs whose defs and/or uses are in well-structured loops must be included in the paths to be constructed.

Figure 11 depicts the types of du-pairs which have to be considered. Figure 11(a) is an example of PEEFSMs that can be generated at Step.3 of the transformation algorithm. If the guard variable of well-structured loops  $\ell_1$  and  $\ell_2$  is  $x$ ,  $\ell_1$  and  $\ell_2$  have  $u_x(\ell_1)$  and  $d_x(\ell_1)$ , and  $u_x(\ell_2)$  and  $d_x(\ell_2)$ , respectively, where  $d_x(t)$  and  $u_x(t)$  are a def and a use of a variable  $x$  in a transition (or a path)  $t$  respectively. When an unconditional transition  $t_1^o$  originating from  $s_1$ , the state of  $\ell_1$  and  $\ell_2$ , has  $u_x(t_1^o)$ , we have to consider du-pairs  $(d_x(\ell_1), u_x(t_1^o))$  and  $(d_x(\ell_2), u_x(t_1^o))$ . This type of du-pair, called Type A, is considered by Step.3 of the algorithm.

Every unconditional path  $T_1$  terminating at  $s_1$  has  $d_x(T_1)$  and there are feasible du-pairs  $(d_x(T_1), u_x(\ell_1))$  and  $(d_x(T_1), u_x(\ell_2))$ . The first requirement of Step.4 handles this type of du-pair, which is called Type B.

Requirement 2 of Step.4 introduces paths that allow the data definitions from some  $l_i$  to propagate onto uses through the inclusion of a path ending in  $l_i$ .

The repetition of a well-structured loop yields feasible du-pairs such as  $(d_x(\ell_1), u_x(\ell_1))$  and  $(d_x(\ell_2), u_x(\ell_2))$ . This type of du-pair, called Type D, is considered by requirement 3 of Step.4. Finally, a combined traversal of two different well-structured loops may also yield feasible du-pairs such as  $(d_x(\ell_1), u_x(\ell_2))$  and  $(d_x(\ell_2), u_x(\ell_1))$ . Requirements 4 and 5 of Step.4 consider this type of du-pairs, which is called Type E. Figure 12 shows an example of the transformation.

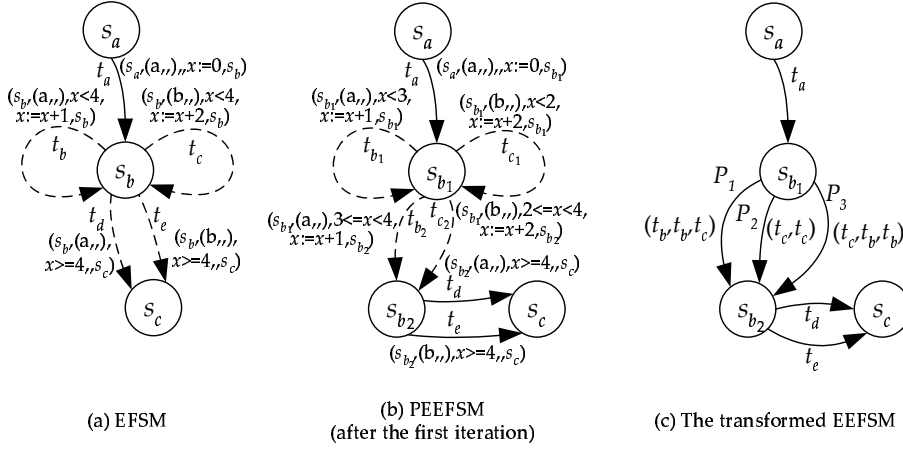


Fig. 12. An example for Step 4

The PEEFSM generated from the NF-EFSM given in Figure 12(a), with the reasonable termination condition, is shown in Figure 12(b). The state from which well-structured loops start is  $s_{b_1}$ . We have one unconditional path  $T_1 = t_a$  going to  $s_{b_1}$ , and  $s_{b_1}$  is not split. Therefore, we only have to consider the transformation of well-structured loops. We skip Step.3 because there is no unconditional transition originating from  $s_{b_1}$ . At Step.4, we consider unconditional paths going from  $s_{b_1}$  to  $s_{b_2}$ . Some of the paths to be added must start with and end with  $t_b$  and  $t_c$  and they must have subpaths  $(t_b, t_b)$  and  $(t_c, t_c)$ . Since the postcondition of  $t_a t_{b_1}$  (which is  $x = 1$ ) implies the disjunction of the preconditions of  $t_{c_1}$  and  $t_{c_2}$  (which is  $x < 4$ ), it is necessary to have subpath  $(t_b, t_c)$ . Similarly, it is necessary to have subpath  $(t_c, t_b)$ . Accordingly, we added three unconditional paths  $P_1, P_2$ , and  $P_3$  to generate a final EEFSM as shown in Figure 12(c).

Where the conditional transitions have simple arithmetic operations, a minimal number of paths satisfying those requirements can easily be constructed. After the final transformed EEFSM is built, test cases satisfying the all-uses criterion can be generated in a straightforward manner. In the transformed

EEFSM, we may have more states and transitions but still have the same number of distinct du-pairs. Some test cases generated from the transformed EEFSM may be longer than the minimized test cases generated from the complete EEFSM because in the transformed EEFSM, some fixed paths are used. However, the difference in length between the two is less than the number of the transitions constructing the fixed path and the number of test cases is identical.

### 6.3 An example

We generate test cases satisfying the all-uses criterion for the PEEFSM of the *Initiator* process shown in Figure 9. The PEEFSM has 10 conditional transitions which were well-structured loops in the NF-EFSM. Using the transformation algorithm given in the previous subsection, those conditional transitions are transformed to unconditional paths as follows. Transitions  $t_{3a}$  and  $t_{3b}$ , starting from the state  $wait_1$  are transformed to a path  $(t_{3a}, t_{3b}, t_{3c}, t_{3d})$  if the only unconditional path  $T_1 = t_1$  going to  $wait_1$  is considered. Note that every path going to  $sending_2$  sets the relevant control variable *counter* to 0. Then we construct the minimal number of paths satisfying the requirements as follows. Although there are unconditional transitions  $t_{14b}$  and  $t_{62a}$  starting from  $sending_2$ , there is no path starting from that state where the guard variable *counter* is used without being re-initialized previously. Therefore, we added no path at Step.3. Both  $t_{7b}$  and  $t_{9b}$  have to be executed three times to satisfy the preconditions of  $t_{7d}$  and  $t_{9d}$  respectively. At Step.4, therefore, the paths have to be composed of the concatenation of four transitions by combining those transitions. At least one of the paths must start with  $t_7$  and at least one must start with  $t_9$ . At least one path must end with  $t_7$  and at least one must end with  $t_9$ . The paths must also contain each subpath composed of the concatenation  $(t_7, t_7), (t_9, t_9), (t_7, t_9)$ , and  $(t_9, t_7)$ . We construct two unconditional paths  $(t_{9a}, t_{9b}, t_{7a}, t_{7b})$  and  $(t_{7c}, t_{9c}, t_{7d}, t_{9d})$  for those conditional transitions. Note that here the labels for the copies of  $t_7$  and  $t_9$  are not intended to correspond to those in Figure 9. For the transitions  $t_{7a}, t_{9a}, t_{7c}$ , and  $t_{9c}$ , we construct two unconditional paths  $(t_{9e}, t_{9f}, t_{7e}, t_{7f})$  and  $(t_{7g}, t_{9g}, t_{7h}, t_{9h})$  similarly. The final transformed EEFSM of *Initiator* process is shown in Figure 13.

All the feasible definition-clear paths for all the du-pairs of the NF-EFSM are derived as shown in Table 2. Those for the input parameters are left out for simplicity because they are defined and used in the same transition.

From the derived definition-clear paths for all the du-pairs, a set of feasible complete paths satisfying the all-uses criterion in the transformed EEFSM is generated as shown in Table 3.

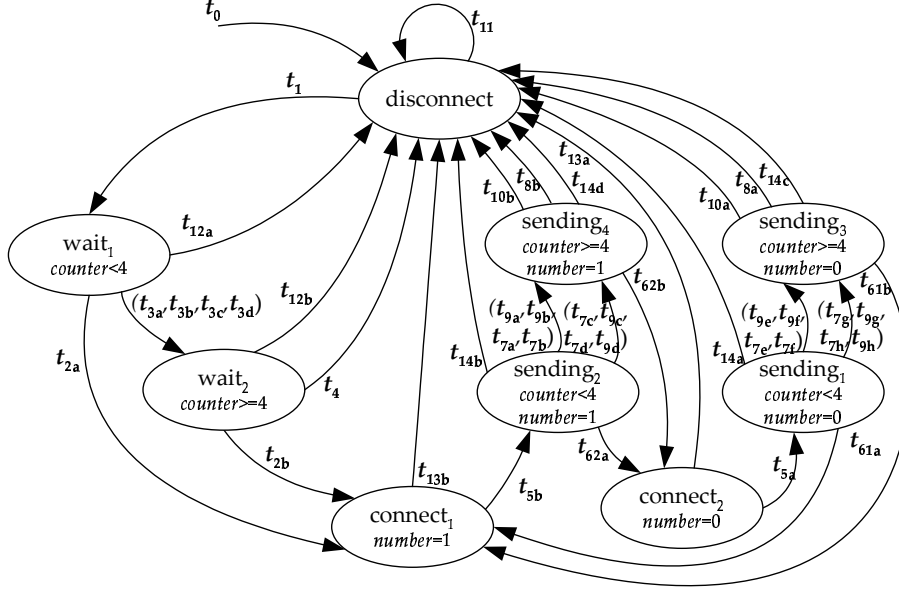


Fig. 13. The transformed EEFSM of *Initiator* process for test generation satisfying all-uses criterion

## 7 Conclusions

This paper has considered the problem of testing a state-based system based on a specification in a formal language. The approach applied has two phases. In the first phase the specification is transformed into a normal form extended finite state machine (NF-EFSM). This phase has been developed for SDL. In the second phase, the NF-EFSM is transformed in order to reduce or eliminate the infeasible path problem in order to aid testing. Splitting the process into these two phases aids generality: in order to extend the approach to some other specification language it is sufficient to define a mapping from that language to NF-EFSMs.

When the output of the second phase is an Expanded EFSM (EEFSM), all paths in this EEFSM are feasible. Test generation may then be based around choosing an appropriate set of paths which guarantee that the test criterion is satisfied, and then finding test data to exercise these paths.

In some cases the EEFSM will be too large and instead the second phase terminates with a Partially Expanded EFSM (PEEFSM). Where this is the case, test generation is more complex. However, the PEEFSM may be further transformed on the basis of the test criterion used: the further expansion is targeted at elements of the test criterion that are not currently satisfiable using unconditional transitions. This paper has given such a transformation algorithm, for the all-uses criterion, that operates under certain conditions. Future work will consider alternative conditions and corresponding transformation algorithms.

## References

- [1] J.Dick and A. Faive, Automating the generation and sequencing of test cases from model-based specifications. *FME'93, First International Symposium on Formal Methods in Europe.*, Odense, Denmark, April 19–23, 1993, pp.268–284.
- [2] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, and M. Schmitt, Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. *in proceedings of the 8th SDL Forum*, Evry, France, 1997.
- [3] German National Research Center for Information Technology, ESPRESS: Engineering of safety-critical embedded systems. GMD FIRST Research Project, <http://www.first.gmd.de/org/espres.html>.
- [4] D. Harel, Statechart: A visual formalism for complex systems. *Science of computer programming*, (8):231–274, 1987.
- [5] O. Henniger, A. Ulrich, and H. Konig, Transformation of Estelle modules aiming at test case generation. *Proceedings of the 8th IFIP International Workshop on Protocol Test Systems*. Evry, France, 1995. Chapman & Hall.
- [6] R.M. Hierons, Testing from a Z specification. *Journal of Software Testing, Verification and Reliability.*, 7(1) (1997) 268–284.
- [7] R.M. Hierons, S. Sadeghipour, and H. Singh, Testing a system specified using Statecharts and Z. *Information and Software Technology.*, 43 (2001) 137–149.
- [8] R.M. Hierons, T.-H. Kim, and H. Ural, Expanding an Extended Finite State Machine to Aid Testability. *COMPSAC 02.*, 2002.
- [9] D. Hogrefe, OSI formal specification case study: the Inres protocol and service. *Technical Report IAM-91-012*, University of Bern, 1991. 5.
- [10] International Organization for Standardization, Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). IS 8825 (November 1987). First Edition.
- [11] ITU, ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunications Union, Geneva, Swizerland, 1999.
- [12] C. B. Jones, Systematic Software Development Using VDM. 2nd ed., Prentice Hall, Hemel Hempstead, U.K. 1990.
- [13] T. H. Kim, I. S. Hwang, M. S. Jang, S. W. Kang, J. Y. Lee and S. B. Lee, Test Case Generation of a Protocol by a Fault Coverage Analysis. *ICOIN-12.*, Tokyo, Japan. (1998)
- [14] C. Meudec, Automatic Generation of Software Test Cases From Formal Specifications. PhD thesis, The Queen's University of Belfast, 1998.

- [15] S. Rapps and E. J. Weyuker, Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, SE-11(4):367–375, April 1985.
- [16] J. R. Shoenfield, *Mathematical Logic*. Addison-Wesley, Reading, Ma., 1967.
- [17] H. Singh, M. Conrad, and S. Sadeghipour, Test Case Design Based on Z and the Classification-Tree Method. *First IEEE International Conference on Formal Engineering Methods.*, Hiroshima, Japan, 1997, pp.81–90.
- [18] J.M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, United Kingdom, 1988.
- [19] J.M. Spivey, *The Z Notation: A Reference Manual*. 2nd ed., Prentice Hall, New York 1992.
- [20] H. Ural, K. Saleh, and A. Williams, Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications.*, 23 (2000) 609–627.
- [21] M. U. Uyar and A. Y. Duale Resolving Inconsistencies in EFSM Modeled Specifications. *IEEE Military Communications Conf. (MILCOM).*, Atlantic City, NJ, October 1999.

Table 2  
The du-pairs in the transformed EEFSM of *Initiator* process

| Variable       | Defined  | Used     | Def-clear Path  | Variable | Defined | Used     | Def-clear Path                     |
|----------------|----------|----------|---|----------|---------|----------|------------------------------------|
| <i>counter</i> | $t_1$    | $t_3$    | $t_1, t_{3a}$   | $T$      | $t_1$   | $t_2$    | $t_1, t_{2a}$                      |
| <i>counter</i> | $t_3$    | $t_3$    | $t_{3a}, t_{3b}$  | $T$      | $t_1$   | $t_3$    | $t_1, t_{3a}$                      |
| <i>counter</i> | $t_3$    | $t_4$    | $t_{3d}, t_4$   | $T$      | $t_1$   | $t_{12}$ | $t_1, t_{12a}$                     |
| <i>counter</i> | $t_5$    | $t_7$    | $t_{5b}, t_{7c}$  | $T$      | $t_3$   | $t_2$    | $t_{3d}, t_{2b}$                   |
| <i>counter</i> | $t_5$    | $t_9$    | $t_{5a}, t_{9a}$  | $T$      | $t_3$   | $t_3$    | $t_{3a}, t_{3b}$                   |
| <i>counter</i> | $t_7$    | $t_7$    | $t_{7a}, t_{7b}$  | $T$      | $t_3$   | $t_4$    | $t_{3d}, t_4$                      |
| <i>counter</i> | $t_7$    | $t_8$    | $t_{7b}, t_{8b}$  | $T$      | $t_3$   | $t_{12}$ | $t_{3d}, t_{12b}$                  |
| <i>counter</i> | $t_7$    | $t_9$    | $t_{7c}, t_{9c}$  | $T$      | $t_5$   | $t_{61}$ | $t_{5a}, t_{61a}$                  |
| <i>counter</i> | $t_7$    | $t_{10}$ | $t_{7b}, t_{10b}$   | $T$      | $t_5$   | $t_{62}$ | $t_{5b}, t_{62a}$                  |
| <i>counter</i> | $t_9$    | $t_7$    | $t_{9b}, t_{7a}$  | $T$      | $t_5$   | $t_7$    | $t_{5b}, t_{7c}$                   |
| <i>counter</i> | $t_9$    | $t_8$    | $t_{9d}, t_{8b}$  | $T$      | $t_5$   | $t_9$    | $t_{5b}, t_{9a}$                   |
| <i>counter</i> | $t_9$    | $t_9$    | $t_{9a}, t_{9b}$  | $T$      | $t_5$   | $t_{14}$ | $t_{5b}, t_{14b}$                  |
| <i>counter</i> | $t_9$    | $t_{10}$ | $t_{9d}, t_{10b}$   | $T$      | $t_7$   | $t_{61}$ | $t_{7f}, t_{61b}$                  |
| <i>number</i>  | $t_2$    | $t_5$    | $t_{2a}, t_{5b}$  | $T$      | $t_7$   | $t_{62}$ | $t_{7b}, t_{62b}$                  |
| <i>number</i>  | $t_2$    | $t_{62}$ | $t_{2a}, t_{5b}, t_{62a}$                                 | $T$      | $t_7$   | $t_7$    | $t_{7a}, t_{7b}$                   |
| <i>number</i>  | $t_2$    | $t_7$    | $t_{2a}, t_{5b}, t_{7c}$                                  | $T$      | $t_7$   | $t_8$    | $t_{7b}, t_{8b}$                   |
| <i>number</i>  | $t_2$    | $t_8$    | $t_{2a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{8b}$  | $T$      | $t_7$   | $t_9$    | $t_{7c}, t_{9c}$                   |
| <i>number</i>  | $t_2$    | $t_9$    | $t_{2a}, t_{5b}, t_{7c}, t_{9c}$                          | $T$      | $t_7$   | $t_{10}$ | $t_{7b}, t_{10b}$                  |
| <i>number</i>  | $t_{61}$ | $t_5$    | $t_{61a}, t_{5b}$   | $T$      | $t_7$   | $t_{14}$ | $t_{7b}, t_{14d}$                  |
| <i>number</i>  | $t_{61}$ | $t_{62}$ | $t_{61a}, t_{5b}, t_{62a}$                                | $T$      | $t_9$   | $t_{61}$ | $t_{9h}, t_{61b}$                  |
| <i>number</i>  | $t_{61}$ | $t_7$    | $t_{61a}, t_{5b}, t_{7c}$                                 | $T$      | $t_9$   | $t_{62}$ | $t_{9d}, t_{62b}$                  |
| <i>number</i>  | $t_{61}$ | $t_8$    | $t_{61a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{8b}$ | $T$      | $t_9$   | $t_7$    | $t_{9b}, t_{7a}$                   |
| <i>number</i>  | $t_{61}$ | $t_9$    | $t_{61a}, t_{5b}, t_{9a}$                                 | $T$      | $t_9$   | $t_8$    | $t_{9d}, t_{8b}$                   |
| <i>number</i>  | $t_{62}$ | $t_5$    | $t_{62a}, t_{5a}$   | $T$      | $t_9$   | $t_9$    | $t_{9a}, t_{9b}$                   |
| <i>number</i>  | $t_{62}$ | $t_{61}$ | $t_{62a}, t_{5a}, t_{61a}$                                | $T$      | $t_9$   | $t_{10}$ | $t_{9d}, t_{10b}$                  |
| <i>number</i>  | $t_{62}$ | $t_7$    | $t_{62a}, t_{5a}, t_{7g}$                                 | $T$      | $t_9$   | $t_{14}$ | $t_{9d}, t_{14d}$                  |
| <i>number</i>  | $t_{62}$ | $t_8$    | $t_{62a}, t_{5a}, t_{7g}, t_{9g}, t_{7h}, t_{9h}, t_{8a}$ | $p$      | $t_0$   | $t_1$    | $t_0, t_1$                         |
| <i>number</i>  | $t_{62}$ | $t_9$    | $t_{62a}, t_{5a}, t_{9e}$                                 | $p$      | $t_0$   | $t_3$    | $t_0, t_1, t_{3a}$                 |
| <i>olddata</i> | $t_5$    | $t_7$    | $t_{5b}, t_{7c}$  | $p$      | $t_0$   | $t_5$    | $t_0, t_1, t_{2a}, t_{5b}$         |
| <i>olddata</i> | $t_5$    | $t_9$    | $t_{5b}, t_{7c}, t_{9c}$                                  | $p$      | $t_0$   | $t_7$    | $t_0, t_1, t_{2a}, t_{5b}, t_{7c}$ |
|                |          |          |   | $p$      | $t_0$   | $t_9$    | $t_0, t_1, t_{2a}, t_{5b}, t_{9a}$ |

Table 3

A set of complete paths satisfying all-uses criterion in the transformed EEFSM of *Initiator* process

---

|     |   |
|-----|---|
| P1  | $t_0, t_1, t_{12a}$   |
| P2  | $t_0, t_1, t_{3a}, t_{3b}, t_{3c}, t_{3d}, t_4$   |
| P3  | $t_0, t_1, t_{3a}, t_{3b}, t_{3c}, t_{3d}, t_{2b}, t_{13b}$   |
| P4  | $t_0, t_1, t_{3a}, t_{3b}, t_{3c}, t_{3d}, t_{12b}$   |
| P5  | $t_0, t_1, t_{2a}, t_{5b}, t_{14b}$   |
| P6  | $t_0, t_1, t_{2a}, t_{5b}, t_{9a}, t_{9b}, t_{7a}, t_{7b}, t_{8b}$                                    |
| P7  | $t_0, t_1, t_{2a}, t_{5b}, t_{9a}, t_{9b}, t_{7a}, t_{7b}, t_{10b}$                                   |
| P8  | $t_0, t_1, t_{2a}, t_{5b}, t_{9a}, t_{9b}, t_{7a}, t_{7b}, t_{14d}$                                   |
| P9  | $t_0, t_1, t_{2a}, t_{5b}, t_{9a}, t_{9b}, t_{7a}, t_{7b}, t_{62b}, t_{13a}$                          |
| P10 | $t_0, t_1, t_{2a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{8b}$                                    |
| P11 | $t_0, t_1, t_{2a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{10b}$                                   |
| P12 | $t_0, t_1, t_{2a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{62b}, t_{13a}$                          |
| P13 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{7g}, t_{9g}, t_{7h}, t_{9h}, t_{8a}$                   |
| P14 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{7g}, t_{9g}, t_{7h}, t_{9h}, t_{61b}, t_{13b}$         |
| P15 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{9e}, t_{9f}, t_{7e}, t_{7f}, t_{61b}, t_{13b}$         |
| P16 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{61a}, t_{5b}, t_{62a}, t_{5a}, t_{14}$                 |
| P17 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{61a}, t_{5b}, t_{7c}, t_{9c}, t_{7d}, t_{9d}, t_{8b}$  |
| P18 | $t_0, t_1, t_{2a}, t_{5b}, t_{62a}, t_{5a}, t_{61a}, t_{5b}, t_{9e}, t_{9f}, t_{7e}, t_{7f}, t_{10b}$ |

---