Expanding an Extended Finite State Machine to aid Testability

R. M. Hierons
Brunel University,
Uxbridge, Middlesex
UB8 3PH, UK
rob.hierons@brunel.ac.uk

T.-H. Kim
University of Ottawa
Ottawa, Ontario
K1N 6N5, Canada
taehyong@site.uottawa.ca

H. Ural
University of Ottawa
Ottawa, Ontario
K1N 6N5, Canada
ural@site.uottawa.ca

Abstract

The problem of testing from an extended finite state machine (EFSM) is complicated by the presence of infeasible paths. This paper considers the problem of expanding an EFSM in order to bypass the infeasible path problem. The approach is developed for the specification language SDL but, in order to aid generality, the rewriting process is broken down into two phases: producing a normal form EFSM (NF-EFSM) from an SDL specification and then expanding this NF-EFSM.

keywords: *extended finite state machine, testability, infeasible paths.*

1 Introduction

Testing is a vital but expensive part of the software verification process. While automation may reduce the cost of testing, automation must be based on some source of information. One 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 modeled or specified using a state-based language such as SDL [6] or Statecharts [3]. A specification in one of these forms may act as the basis for automating or semi-automating testing[2, 7]. Such a specification is rewritten to form an extended finite state machine (EFSM) and tests generated from this EFSM.

The process of generating tests from an EFSM may be split into two steps: first find a set of paths that, between them, satisfy the test criterion and then produce test sequences for each of these path. Thus test generation may 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 approach is developed for SDL. However, potentially it might be extended to any model-based language such as Z or state-based specification language such as Statecharts. This paper extends the work of [5] on the refinement of an EFSM for the generation of executable tests.

This paper is organized as follows. Section 2 explains how SDL specifications may be represented as EFSMs. Section 3 then defines a normal form EFSM (NF-EFSM). The expansion procedure, which forms the core of this paper, is proposed in Section 4. The procedure is composed of two phases: building an NF-EFSM and expanding this to improve testability. The use of an NF-EFSM aids generality: once a specification has been rewritten to this form the expansion procedure may be applied. Section 5 applies the procedure to an example. Section 6 considers the problem of generating tests from the expanded EFSM. Finally, in Section 7, conclusions are drawn.

2 Formal Methods and EFSMs

Formal methods are mathematical techniques for specifying complex systems. Most formal methods focus on the sequential or concurrent behaviour of the system and such specifications can be seen as a single EFSM or multiple EFSMs communicating with each other.

SDL is a specification and description language standardized by ITU. An SDL specification is graphical and symbol-based and can be seen as a set of EFSMs communicating with each other.

While the sequential behaviour of most formal specifications can be considered as an EFSM, a transition may contain conditional statements. The conditions control which behaviour is applied. Normally each of these behaviours should be tested. Such a transition may be replicated to give one transition for each behaviour, in order to ensure that each behaviour is tested. 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.

A number of test criteria have been considered for testing against an EFSM. These criteria typically fall into one of two categories: control flow and data flow criteria. Control flow criteria test the structure of the implementation. These criteria are typically based on finite state machine techniques. In contrast data flow criteria test ways in which data may be transferred.

Typically data flow criteria consider definitions and uses of variables. Given a variable v, a definition of v is some assignment of a value to v. A use of v is an assignment or output that references v (c-use) or a guard that references v (p-use). A definition n_1 of v may propagate onto a use n_2 of v through a definition clear path for v: a path from n_1 to n_2 that contains no definitions of v between n_1 and n_2. Then n_1, n_2 forms a du-pair for v. Data flow criteria are typically expressed in terms of du-pairs. In particular, the all-uses test criterion [8] is satisfied if: for every variable v, and every du-pair n_1, n_2 for v, some test follows a definition clear path for v from n_1 to n_2. Note that where the use n_2 is a p-use, n_2 is a transition not a state.

3 Overview of the Proposed Approach

This paper focuses on the problem of producing an expanded EFSM given a specification of a deterministic sequential system in SDL. 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.

The initial normalization phase of a specification varies according to its formal method but the expansion phase is common for any specification. A normal form EFSM is defined as follows.

Definition 1 A *normal form extended finite state machine (NF-EFSM) M* is defined by the tuple $(S, s_0, V, v_0, P, I, O, T)$ in which: S is the set of logical states; $s_0 \in S$ is the initial state; V is the set of internal variables; v_0 gives the initial values of the internal variables; P is the set of input parameters; I is the set of input declarations; O is the set of output declarations; and T is the set of transitions. The label of transition $t \in T$ is the tuple (s_s, g, op, s_f) in which s_s is the start state of t, g is the guard and can be represented as $f_L(I, V, P)$, where $f_L(\cdot)$ is the logical expression, op is the operation which is composed of only output statements and assignment statements, and s_f is the final state of t.

External events that may trigger transitions are represented as input declarations. Input parameters are the attributes or parameters of those external events. V contains all the variables that occupy memory in the system. Among the variables in V, we call those used in guards *control variables*.

$\mathcal{D}$ denotes the domain constructed from the control variables in V and Λ the domain constructed from the input parameters in P which are related to control variables in V. In addition, we will use '*domain* of a state' as a subset of $\mathcal{D}$ allowed at the state.

We assume an NF-EFSM is deterministic, strongly connected, minimized, and completely specified. The motivation of an NF-EFSM is as follows. First an NF-EFSM is independent of the syntax of the specification language used. Second, an NF-EFSM is a suitable form for test generation, because every operation of a transition in an NF-EFSM represents a single behaviour. Finally, most of the existing methods for test generation can be applied directly to an NF-EFSM even if we don't expand it.

4 The Expansion Procedure

This section describes the procedure that expands a sequential SDL specification to form an Expanded EFSM (EEFSM) or a Partially Expanded EFSM (PEEFSM). This algorithm is iterative and thus avoids the introduction of non-determinism that may result from state splitting [5].

4.1 Phase I: Building an NF-EFSM

A process diagram in SDL is an EFSM. A transition from one logical state to another is described in a series of symbols. The guard of a transition is defined using 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 make an NF-EFSM, the process diagram should have only a single decision symbol for a transition. If an operation has complex elements such as multiple decision symbols, cyclic paths, timer operations, saves, procedure calls, etc, it can be flattened using various techniques [9].

It may be useful to apply domain propagation[1]. Here an operator may be partitioned so that its behaviour in a subdomain is considered to be uniform. For example, an operation that returns the absolute value of a variable x may be split into two cases: one where $x \geq 0$ and one where $x < 0$.

4.2 Phase II: Expansion

First, we introduce some notation and functions. We restrict the meanings of the precondition and the postcondition of a transition in the algorithm as follows. The guard g_i of a transition t_i is split into the precondition, usually denoted by P_i, and the parameter condition, λ_i, namely $g_i = P_i \wedge \lambda_i$. The parameter condition of a transition is the logical expression composed of all the atomic predicates that mention an input or input parameter. The precondition of a transition is the remaining part of the guard, the expression composed of the predicates that mention only control variables. The unary *dom* operator generates from a logical expression the corresponding subdomain in $\mathcal{D}$ while the unary *cond* operator generates from a subdomain of $\mathcal{D}$ the corresponding logical expression. The postcondition function of t_i, usually denoted by $Q_i(\cdot) : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\Lambda) \rightarrow \mathcal{P}(\mathcal{D})$, is the function that derives a subdomain in $\mathcal{D}$, according to the operation op_i of t_i, given subdomains in $\mathcal{D}$ and Λ respectively. $d(\cdot) : S \rightarrow \mathcal{D}$ 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. We say t_i is *unconditional* if $dom\ P_i \supseteq d(s_{ST}(t_i))$; otherwise, it is *conditional*, where P_i is the precondition of t_i. The algorithm assumes that all the postcondition functions and their inverse functions can be evaluated symbolically in any domain considered.

4.2.1 Algorithm

Step 0: If the guard g_i of a transition t_i is not in the form $P_i \wedge \lambda_i$, split the transition into transitions $t_{i_1}, \cdots, t_{i_n}$ whose operations are the same as that of t_i and whose guards are $g_{i_1}, \cdots, g_{i_n}$ satisfying $g_{i_1} = P_{i_1} \wedge \lambda_{i_n}, \cdots, g_{i_n} = P_{i_1} \wedge \lambda_{i_n}$, and $g_i = g_{i_1} \vee \cdots \vee g_{i_n}$.
Step 1: Given state s_s, if the transitions $t_1, t_2, \cdots, t_n$ starting from s_s are conditional and have preconditions $P_1, P_2, \cdots, P_n$ respectively then partition the domain of s_s as follows. Each subdomain, $\mathcal{P}^s_X$, $X \subseteq \{1, \ldots, n\}$, $X \neq \{\}$ is given by

$$\mathcal{P}^s_X = dom\ ((\wedge_{i \in X} P_i) \wedge (\wedge_{i \notin X} \neg\ P_i)).$$

The number of subdomains is at most $2^n - 1$ but may be fewer because some may be empty. For example, if an operation at a state s_s is rewritten as $\bigvee_{1 \leq i \leq 3}(P_i \wedge Q_i)$, a partition of the domain of s_s is:

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

If the final non-empty disjoint subdomains are $\mathcal{P}^s_1, \cdots, \mathcal{P}^s_m (m \leq 2^n - 1)$, split state s_s to $s_{s_1}, \cdots, s_{s_m}$ whose domains are $\mathcal{P}^s_1, \cdots, \mathcal{P}^s_m$, respectively.

If this is the first iteration, repeat this step for all the states which have conditional transitions. After the first iteration, the state to be split may be selected arbitrary or purposely.
Step 2: Rearrange transitions related to the split states. If a state s_i is split into $n (\geq 2)$ states, $s_{i_1}, \cdots, s_{i_n}$, remove each transition t_j going from or to the state s_i. Then, for each removed transition t_j going from s_i to a state $s_f (\neq s_i)$, make n temporary transitions, one from each $s_{i_k (1 \leq k \leq n)}$ to s_f, whose labels are the same as t_j. For each removed transition t_j going to s_i from a state $s_s (\neq s_i)$, make n temporary transitions, one from s_s to each $s_{i_k (1 \leq k \leq n)}$, whose labels are the same as t_j. For each removed transition t_j going from and to s_i make n^2 temporary transitions, one 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 t_j.
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 $\mathcal{P}^{s_{ST}(t_i)}_X$ for some X: $d(s_{ST}(t_i)) \subseteq dom\ P_i$ or $d(s_{ST}(t_i)) \cap dom\ P_i = \varnothing$. Therefore, for each t_i, make t_i permanent or discard it depending on the following cases:
Case A: $dom\ P_i \cap d(s_{ST}(t_i)) = \varnothing$ or $Q_i(d(s_{ST}(t_i)), dom\ \lambda_i) \cap d(s_{FN}(t_i)) = \varnothing$. Discard t_i.
Case B: $dom\ P_i \supseteq d(s_{ST}(t_i))$ and $Q_i(d(s_{ST}(t_i)), dom\ \lambda_i) \subseteq d(s_{FN}(t_i))$. Make t_i unconditional with guard λ_i.
Case C: $dom\ P_i \supseteq d(s_{ST}(t_i))$, $Q_i(d(s_{ST}(t_i)), dom\ \lambda_i) \nsubseteq d(s_{FN}(t_i))$ and $Q_i(d(s_{ST}(t_i)), dom\ \lambda_i) \cap d(s_{FN}(t_i)) \neq \varnothing$:

- make t_i unconditional, with guard $\lambda_i \wedge$ *cond* Q_i^{-1} $(d(s_{FN}(t_i)))$, if $dom\ P'_i \supseteq d(s_{ST}(t_i))$.
- make t_i conditional, with guard $\lambda_i \wedge$ *cond* Q_i^{-1} $(d(s_{FN}(t_i)))$, if $dom\ P'_i \nsupseteq d(s_{ST}(t_i))$.

Here P'_i is the new precondition of t_i according to its new guard, P_i and λ_i are the precondition and the parameter condition of t_i respectively, and $Q_i(\cdot)$ is the postcondition function of 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 one of Conditions A, B, and C is satisfied, terminate; otherwise, return to Step 1.
Condition A (Complete expansion): There are no conditional transitions.
Condition B (Sufficient expansion): There are conditional transitions but there exists some set P of paths, that correspond to tests that satisfy the test criterion, such that each path from P contains only unconditional transitions.
Condition C (Termination due to scale): Neither of Conditions A and B is satisfied but further expansion is considered to be impractical.

Where the algorithm terminates using condition A the resultant EFSM is an expanded EFSM (EEFSM); otherwise it is a partial expanded EFSM (PEEFSM).

Note that the choice of state to expand may be crucial and the ideal choice may depend upon the test criterion used. The development of approaches that direct expansion for particular test criteria will form a part of future work.

4.2.2 Justification

The algorithm partitions the domain of each logical state with the preconditions of its conditional transitions. When a state is split, several conditional transitions may be generated by the split state. So, the algorithm may have to split states repeatedly.

The algorithm tries to generate a minimized reachability state machine by keeping states in the reachability tree, on which the behaviour of the system is uniform, as a state in the EEFSM. Thus the EEFSM is at most as large as the reachability tree. Therefore, if the reachability tree is finite, the algorithm will terminate. This is guaranteed if the domain constructed from the control variables is finite. This is a sufficient, but not necessary, condition.

Sometimes it may be reasonable to terminate the process before the test criterion is satisfied using sequences of unconditional transitions (Condition C). In this case feasible paths may be added to allow the test criterion to be satisfied.

5 Example of expansion

We derive a (P)EEFSM from the process diagram of the *Initiator* process of the *Inres* protocol[4] shown in Figure 1.

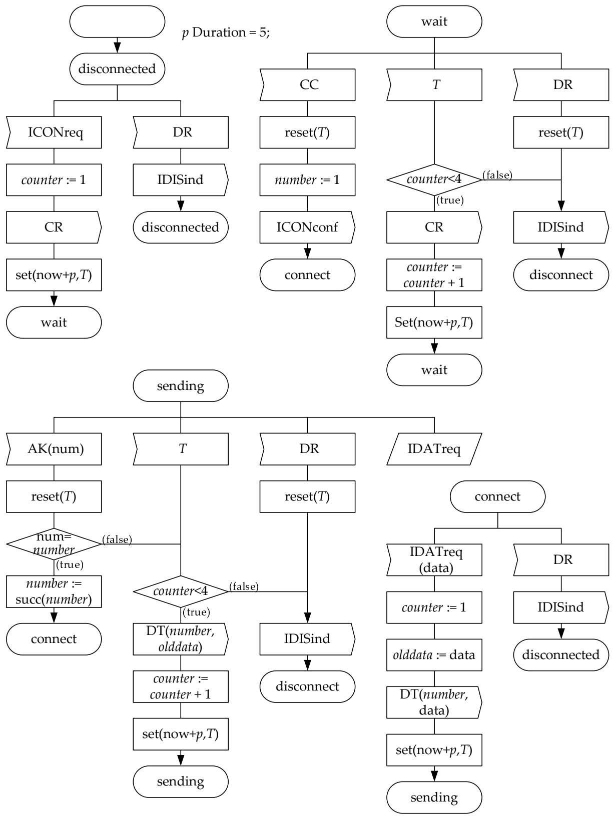
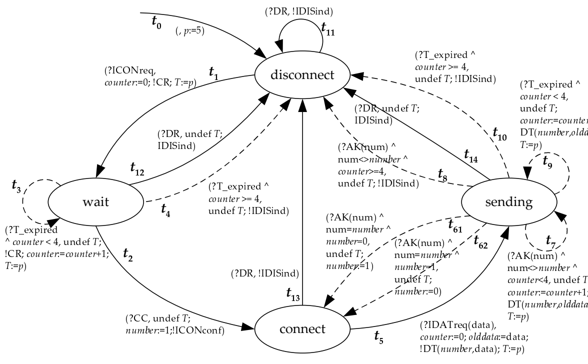


Figure 1. The *Initiator* process

5.1 Phase I

To build the NF-EFSM of *Initiator*, timer operations are flattened as follows. For a timer T, we define a variable T for saving the remaining time to the expiry of timer T. If there are more than two timers, we define a variable *min_timer* for the minimum value of all currently active timeout periods[9]. The timer expiry input of T is changed to the input *T_expired* and the statement 'undef T'. 'Undef' statement of a variable makes the variable considered undefined and the variable is considered to be used in the statement. *Set* of timer T to a duration is converted to the assignment of the duration to variable T, and *reset* of timer T is converted to the statement 'undef T'. It is difficult to flatten *save* operations in general. In this example, a *save* operation is used to keep the user data from being lost. Here, 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 'connect' state. For testing a *save* operation of an input, feasible subpaths may be added to the NF-EFSM as new transitions which start with the transition having the *save* operation and end with transitions whose guard has the input.

The function 'succ' toggles between 0 and 1 for the value of a binary variable. The task $number := \mathrm{succ}(number)$ is flattened to give two behaviours: the result is 1 if $number = 0$ and the result is 0 if $number = 1$. As will be seen in Section 6, this flattening simplifies test generation. The NF-EFSM of *Initiator* process is shown in Figure 2.



t0 (, p:=5)
(?DR, !IDISind) t11
disconnect
(?ICONreq, counter:=0; !CR; T:=p) t1
(?T_expired ^ counter >= 4, undef T; !IDISind)
(?T_expired ^ counter < 4, undef T; counter:=counter+1; DT(number,olddata); T:=p)
(?DR, undef T; IDISind)
t10
(?DR, undef T; IDISind)
t12
(?AK(num) ^ num<>number ^ counter>=4, undef T; !IDISind)
t14
t8
t9
t3
wait
t4
(?T_expired ^ counter >= 4, undef T; !IDISind)
sending
(?T_expired ^ counter < 4, undef T; !CR; counter:=counter+1; T:=p)
t2
(?AK(num) ^ num=number ^ number=0, undef T; number:=1)
t61
(?AK(num) ^ num=number ^ number=1, undef T; number:=0)
t62
t7
(?AK(num) ^ num<>number ^ counter<4, undef T; counter:=counter+1; DT(number,olddata); T:=p)
(?DR, !IDISind)
t13
(?CC, undef T; number:=1;!ICONconf)
connect
t5
(?IDATreq(data), counter:=0; olddata:=data; !DT(number,data); T:=p)

Figure 2. The NF-EFSM of *Initiator*

5.2 Phase II

At Step.0, all guards are in the required form. At Step.1, the domain of state *wait* is partitioned according to the conditions $(counter < 4)$ and $(counter >= 4)$. So, *wait* is split as follows: $wait_1$, defined by $(counter < 4)$; and $wait_2$, defined by $(counter \geq 4)$.

Since this is the first iteration, the domain of state *sending* is also partitioned according to the conditions, $(counter < 4)$, $(counter \geq 4)$, $(number = 0)$, and $(number = 1)$ from transitions $t_{61}, t_{62}, t_7, t_8, t_9$, and t_{10}. The state *sending* is split to four states as follows: $sending_1$

(*counter* $< 4 \wedge$ *number* $= 0$); *sending*$_2$ (*counter* $< 4 \wedge$ *number* $= 1$); *sending*$_3$ (*counter* $\geq 4 \wedge$ *number* $= 0$); and *sending*$_4$ (*counter* $\geq 4 \wedge$ *number* $= 1$).

At Step.3, 18 temporary transitions become unconditional (Case B), 12 become conditional (Case C), and the others are discarded (Case A). At the end of this step, the PEEFSM is that shown in Figure 3. Here, each copy of a transition multiplied by states splitting is given a distinct label in order to aid explanation.

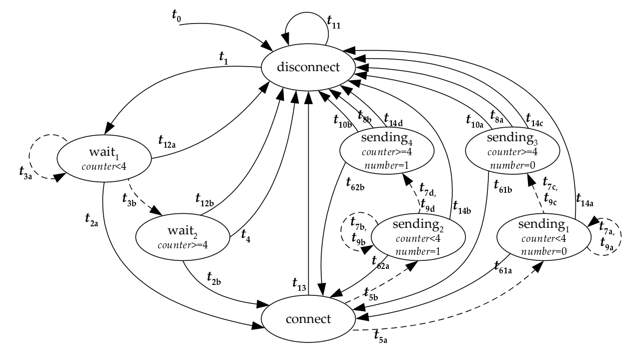


Figure 3. After the first iteration

At Step.4, Conditions A and B are not satisfied and we still have 12 conditional transitions.

At Step.1, on the second iteration, the state *connect* is split as follows: *connect*$_1$ (*number* $= 1$); and *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 is generated as shown in Figure 4.

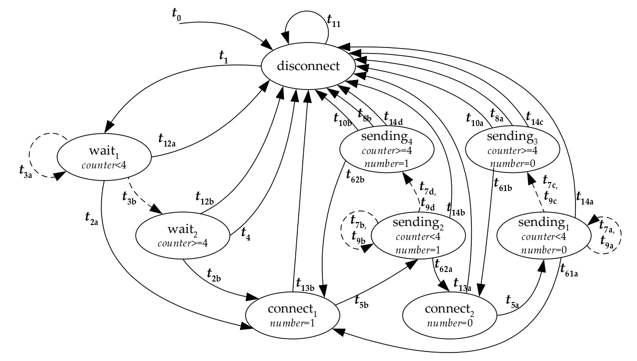


Figure 4. After the second iteration

While it is clear that the expansion process, if allowed, would terminate with an EEFSM, the process will now be stopped here under Condition C. This will allow us to illustrate some of the issues involved in testing from a PEEFSM.

6 Test Generation

This section will consider the problem of generating tests, that satisfies all-uses, from an EEFSM or PEEFSM. A (P)EEFSM can be expanded further to test a specific part of the system [5]. Domain propagation may be used to split some transitions. The expanding algorithm may then be executed to make an equivalent (P)EEFSM.

If expansion terminates under either Condition A or Condition B, test generation is relatively simple: a set of feasible paths is generated and a test sequence produced for each of these. In order to produce a test sequence for a feasible path it is sufficient to determine the path condition and then produce test input that satisfies this path condition.

Now consider the problem of generating test cases satisfying all-uses for the PEEFSM shown in Figure 4. The PEEFSM has 10 conditional transitions which will be replaced by feasible paths that cover the required du-pairs. In this case the problem is simplified by the fact that the states *wait*$_1$, *sending*$_1$, and *sending*$_2$ are only reached by transitions that set *counter* to zero.

The transitions t_{3a} and t_{3b} from *wait*$_1$ are transformed to a path $(t_{3a}, t_{3b}, t_{3c}, t_{3d})$ because it is sufficient to have feasible paths that contain the du-pairs $(d^c_{t_1}, u^c_{t_3})$, $(d^c_{t_3}, u^c_{t_3})$, and $(d^c_{t_3}, u^c_{t_4})$, where c means variable *counter* and (d^x_i, u^x_j) is a du-pair composed of a definition of variable x in transition i and a use of x in transition j.

For the transitions t_{7b}, t_{9b}, t_{7d}, and t_{9d} going from *sending*$_2$, we construct the minimal number of feasible paths satisfying those requirements as follows. Between them, t_{7b} and t_{9b} must be executed three times to satisfy the preconditions of t_{7d} and t_{9d} respectively. Thus the paths added are composed of the concatenation of four of these transitions. The final EEFSM must have feasible paths that contain all the following du-pairs: $(d^n_{t_2}, u^n_{t_7})$, $(d^n_{t_2}, u^n_{t_9})$, $(d^n_{t_2}, u^n_{t_8})$, $(d^c_{t_7}, u^c_{t_7})$, $(d^c_{t_7}, u^c_{t_8})$, $(d^c_{t_7}, u^c_{t_9})$, $(d^c_{t_7}, u^c_{t_{10}})$, $(d^c_{t_9}, u^c_{t_7})$, $(d^c_{t_9}, u^c_{t_8})$, $(d^c_{t_9}, u^c_{t_9})$, and $(d^c_{t_9}, u^c_{t_{10}})$, where n denotes variable *number*. We construct unconditional paths $(t_{9a}, t_{9b}, t_{7a}, t_{7b})$ and $(t_{7c}, t_{9c}, t_{7d}, t_{9d})$ for those du-pairs. For the transitions t_{7a}, t_{9a}, t_{7c}, and t_{9c}, we construct unconditional paths $(t_{9e}, t_{9f}, t_{7e}, t_{7f})$ and $(t_{7g}, t_{9g}, t_{7h}, t_{9h})$. The final transformed EEFSM of *Initiator* process is shown in Figure 5.

Feasible definition-clear paths for all the DU-pairs of the NF-EFSM may now be derived. A set of feasible complete paths satisfying all-uses criterion is shown in Figure 6.

7 Conclusions

This paper has introduced an approach for improving the testability of a state-based specification. The approach has two phases. In the first phase the specification is rewritten

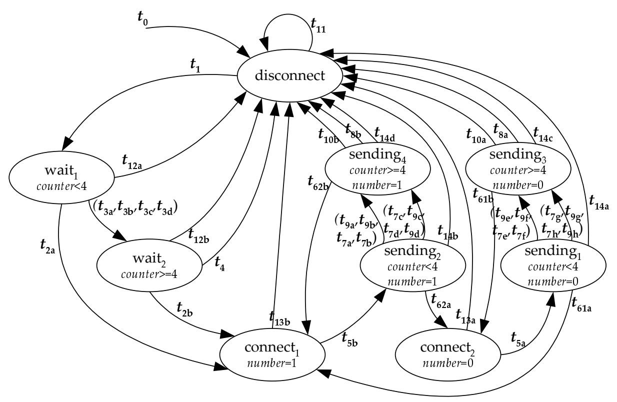


Figure 5. The transformed EEFSM of *Initiator* process for test generation satisfying all-uses criterion

1	12									
1										
1			2	1						
1			12							
1	2	1								
1	2									
1	2				1					
1	2				1					
1	2				2		1			
1	2									
1	2				1					
1	2				2		1			
1	2	2								
1	2	2						1	1	
1	2	2						1	1	
1	2	2	1			2		1		
1	2	2	1							
1	2	2	1							1

Figure 6. A complete set of paths

to form a normal form extended finite state machine (NF-EFSM). This phase has been described for SDL. The NF-EFSM is then refined to form an Expanded EFSM (EEFSM) that has properties that simplify test generation. Splitting the process into these phases aids generality: in order to extend the approach to another specification language it is sufficient to define a mapping from that language to NF-EFSMs.

When the output of the second phase is an EEFSM, all paths in this EEFSM are feasible. In some cases it is not necessary to expand to the EEFSM; the test criterion may be satisfied using feasible paths drawn from a Partially Expanded EFSM (PEEFSM). In each of these cases test generation is based around choosing an appropriate set of paths and then finding test data to exercise these paths. In some cases, due to issues of scale, the expansion may terminate with a PEEFSM before either of these conditions is satisfied. Here, feasible paths may be added to the PEEFSM so that this PEEFSM contains a set of feasible paths that, between them, satisfy the test criterion used.

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] D. Harel, Statechart: A visual formalism for complex systems. *Science of computer programming*, (8):231–274, 1987.

[4] D. Hogrefe, OSI formal specification case study: the Inres protocol and service. *Technical Report* IAM-91-012, University of Bern, 1991. 5.

[5] R.M. Hierons, S. Sadeghipour, and H. Singh, Testing a system specified using Statecharts and Z. *Information and Software Technology*., 43 (2001) 137–149.

[6] ITU, ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunications Union, Geneva, Switzerland, 1999.

[7] C. Meudec, Automatic Generation of Software Test Cases From Formal Specifications. PhD thesis, The Queen's University of Belfast, 1998.

[8] 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.

[9] 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.