On EFSM-based Test Derivation Strategies [1]

Aleksandr Nikitin, Natalia Kushik

***Abstract*—Formal model based test derivation is now widely used in software testing. One of the formal models which is very close to software implementations is the model of an Extended Finite State machine (EFSM). Compared with an FSM the EFSM has predicates for condition representation and context variables. However, when deriving tests for an EFSM very complex reachability and distinguishability problem should be solved. For this reason, when deriving tests from an EFSM a number of FSM slices are used. In this paper, we discuss how to derive a test using an FSM slice with limited number of states and how to represent data in the PC memory for fast generation of such slice. Preliminary experimental results with protocol EFSMs are provided.**

***Index Terms*—Software testing, EFSM, computer representation**

I. Introduction

THE complexity of digital systems and devices increases quickly and software is a usual part of almost each system or device. Thus, ad hoc testing of software implementations now is insufficient and a number of methods for formal model based software testing are proposed and widely used, since model based testing provides tests with the guaranteed fault coverage.

One of formal models which is very close to software implementations is the model of an Extended Finite State machine (EFSM). The EFSM model extends the classic FSM model with input and output parameters, context variables, operations and predicates defined over context variables and input parameters. If specification domains of input parameters and context variables are finite then an EFSM can be unfolded to an equivalent FSM (FSM slice) by simulating its behavior with respect to all possible values of context variables and input vectors [1]. A test suite then is derived from the corresponding equivalent FSM. However, the number of states of such corresponding FSM grows exponentially, and thus, it is necessary to limit the maximal number of states and in this case, the corresponding FSM becomes nondeterministic. In this paper, we show how tests can be derived for such nondeterministic FSM slice. In order to derive tests effectively an efficient computer representation of the complex EFSM model is proposed in this paper.

The structure of the paper is as follows. Section II contains preliminaries and a discussion how tests for an EFSM can be derived when the number of a corresponding FSM is limited. Section III is devoted to EFSM computer representation. Section IV discusses preliminary experimental results with protocol EFSMs while Section V concludes the paper.

II. Preliminaries

The EFSM Model

An extended finite state machine [2] A is a pair (S, T) of a set of states S and a set of transitions T between states from S, such that each transition $t \in T$ is a tuple (s, x, P, op, y, up, s'), where:

$s, s' \in S$ are the initial and final states of a transition;

$x \in X$ is an input, and $D_{inp\text{-}x}$ is the set of possible input vectors, associated with the input x, i.e., each component of an input vector is the value of a corresponding input parameter associated with x;

$y \in Y$ is output, where Y is the set of outputs, and $D_{out\text{-}y}$ is the set of possible output vectors, associated with the output y, i.e. each component of an output vector corresponds to an output parameter associated with y;

P, op, and up are functions, defined over input parameters, and context variables, namely:

$P: D_{inp\text{-}x} \times D_V \rightarrow$ {True, False} is a predicate, where D_V is a set of context vectors $\mathbf{v}$;

$op: D_{inp\text{-}x} \times D_V \rightarrow D_{out\text{-}y}$ is an output parameter update function;

$up: D_{inp\text{-}x} \times D_V \rightarrow D_V$ is a context update function.

As in [2], we use the following definitions.

Given an input x and a vector $\boldsymbol{\rho} \in D_{inp\text{-}x}$, the pair ($x$, $\boldsymbol{\rho}$) and vector from $D_{inp\text{-}x}$, is called a *parameterized input*. A sequence of parameterized inputs is called a *parameterized input sequence*. A context vector $\mathbf{v} \in D_V$ is called a *context* of M. A *configuration* of M is a pair (s, $\mathbf{v}$). Given a parameterized input sequence of an EFSM we can calculate the corresponding parameterized output sequence by simulating the behavior of the EFSM under the input sequence starting from the initial state and initial values of the context variables.

As an example, consider the EFSM E in Figure 2 that corresponds to the C function f presented in Figure 1.

[1] This work is partly supported by FCP grant № 02.514.12.402.

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

Figure 1. The function *f*

Function *f* in Figure 1 returns the maximal integer in the array *a* where *size_a* is the cardinality of *a*. To obtain an EFSM that corresponds to the given C function we first determine the set *S* of states. Let *S* be the set $S = \{s_1, s_2, s_3\}$ where s_1, s_2, s_3 are three different points in the C function. The state s_1 corresponds to the beginning of the function *f*; the state s_2 represents the state of the program before comparing *i* with *size_a*; the program moves to the state s_3 if *i* is less than *size_a*. The set *X* of input consists of the array pointer **a* and of the cardinality *size_a* of *a*. Input **a* is a parameterized input, here *index* (item number) is a parameter. Output $y \in Y$ is not parameterized; it corresponds to the variable *m* that is returned by the function *f*. We also add special input (and output) '*NULL*' to specify cases when program accepts (or returns) no external data. The set *P* of predicates consists of P_1 and P_2: P_1 is *true* if *i* is less than *size_a* while P_2 is *true* if *m* is less than *a*[*i*]. The variable *i* is the context variable. The corresponding EFSM *E* is presented in Figure 2.

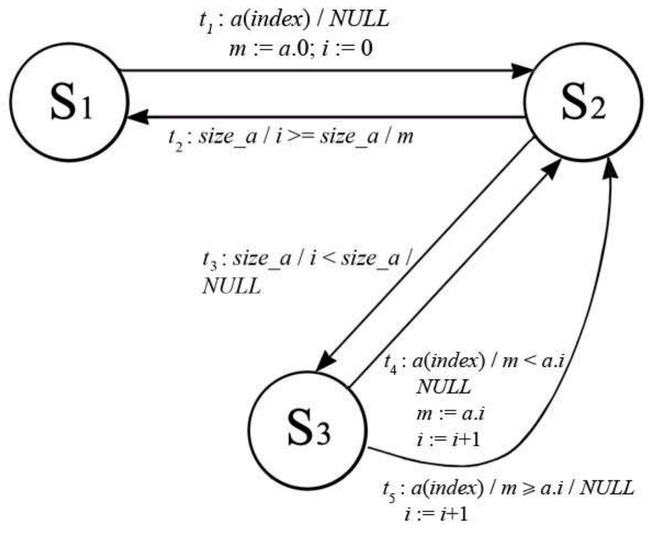
t_1 : a(index) / NULL
m := a.0; i := 0
S1
S2
t_2 : size_a / i >= size_a / m
t_3 : size_a / i < size_a / NULL
t_4 : a(index) / m < a.i / NULL
m := a.i
i := i+1
S3
t_5 : a(index) / m ⩾ a.i / NULL
i := i+1

Figure 2. The EFSM *E*

In this paper, we consider deterministic and complete EFSMs, i.e., for each parameterized input sequence there exists a single parameterized output sequence that is produced by the EFSM for the given input sequence.

Unfolding a given EFSM to an equivalent FSM

Given an EFSM *A*, in order to obtain an equivalent FSM *F* we proceed as follows. Given a state *s* of EFSM *A*, a context vector **v**, an input *x* and vector **ρ** of input parameters, we derive the transition from configuration (*s*, **v**) under input (*x*, **ρ**) in the corresponding FSM *F*. We first determine the outgoing transition (*s*, *x*, *P*, *op*, *y*, *up*, *s´*) from state *s* where the predicate *P* is true for input vector **ρ** and context vector **v**, update the context vector to the vector **v´** according to the assignment *up* of this transition, determine the parameterized output *y***ω** and add the transition (*s***v**, *x***ρ**, *y***ω**, *s´***v´**) to the set of transitions of the FSM. The obtained FSM has the same number of states as the number of different configurations (*s*, **v**) of the EFSM *F* that are reachable from the initial state.

Such unfolding can help to detect transfer, predicate, and assignment faults of the given EFSM *A*. However, it is known that the simulation usually leads to a state explosion problem. That is the reason why the maximal number of states of the FSM *F* is limited by integer *B*, for example. In this case, all the states corresponding to configurations (*s*, **v**) with the numbers that are greater then *B* are marked by a special state DNC (don't care state). Two ways are then appropriate for FSM *F* testing.

a) Transitions with DNC states are deleted from *F* and *F* is tested as a partial FSM [3].

b) FSM *F* is tested as completely specified FSM [4] and then the test suit is "cleaned" by deleting all suffixes of test sequences that lead to the DNC state. However, the fault coverage of such a test suite is still unknown.

As an EFSM model is rather complex, a suitable computer EFSM representation can be of a big help when unfolding and deriving tests automatically.

III. EFSM COMPUTER REPRESENTATION

Computer representation of the EFSM *A* uses the following items kept in the PC memory: a number of states of *A*, an array of parameterized inputs, an array of parameterized outputs, an array of context variables, an array of functions *op* and *up*, an array of integers that are used in functions *op* and *up*, an array of predicates, and an array of transitions. We define a set of structures in the C language for the computer representation of an EFSM.

Structure *input* is used for parameterized (or not parameterized) input representation.

```
struct input
{
char *input_name;
int par_quantity;
char **parameters_names;
float *parameters_values;
};
```

The field *input_name* of the structure *input* captures initial input name while the *par_quantity* field is a number of input parameters. The *parameters_names* array saves initial parameter names and the *parameters_values* array is used for an assignment of the input parameters.

Structure *output* is similar to *input* structure and is used for parameterized (or not parameterized) output representation.

```
struct output
{
char *output_name;
int par_quantity;
char **parameters_names;
float *parameters_values;
};
```

The field *output_name* of the structure *output* captures initial output name; the *par_quantity* field is a number of output parameters. The *parameters_names* array is also used for initial parameters names and the *parameters_values* array keeps an assignment of the output parameters.

We use **structure *variable*** for context variable representation.

```
struct variable
{
char *variable_name;
float value;
};
```

The field *variable_name* of the structure *variable* corresponds to initial name of the variable while value of the context variable is stored in the *value* field.

When simulating behavior of the EFSM A we use integers instead of strings. In other words, we hash inputs, outputs, variables, functions and predicates and use corresponding integer identifiers when deriving tests. We note that such a hashing significantly accelerates the simulation process. Reverse Polish Notation [5] is utilized for faster translation of EFSM predicates and functions into computer representation. That is the reason why the **structure *function*** has two fields.

```
struct function
{
int *rpr;
int rpr_size;
};
```

The Reverse Polish Notation that corresponds to the function is stored in the *rpr* array of *rpr_size* items. The items of the *rpr* array are identifiers of parameterized inputs or context variables. Arithmetic operators are also hashed and stored in the *rpr* array. We use special *rpr* item '–1' to separate operators and operands of the Reverse Polish Notation.

When constructing the Reverse Polish Notation for the predicate P we hash comparison operators and divide an arithmetic expression into two parts: the arithmetic expression that is in the left hand side of the comparison operator is the '*left notation*'. In the right hand side of the comparison operator is the '*right notation*'. Correspondingly, we consider only predicates where left hand and right hand side expressions are separated with one of the following comparison operators {<, >, >=, <=, ==, !=}. Therefore, **structure *predicate*** has four fields.

```
struct predicate
{
int *rpr_left;
int rpr_left_size;
int *rpr_right;
int rpr_right_size;
int sign_op;
};
```

The *rpr_left* array of *rpr_left_size* items corresponds to the 'left notation' while *rpr_left* array of *rpr_left_size* items are used for the 'right notation'. The *sign_op* field corresponds to the comparison operator identifier.

As the EFSM is a pair (S, T) of a set of states S and a set of transitions T, it is necessary to save all the transitions of the set T. Correspondingly we define a **structure *transition*** for $t \in T$.

```
struct transition
{
int s;
int s_prime;
input i;
output o;
int *predicate_numbers;
int *function_numbers;
};
```

Integers s and *s_prime* are initial and final states of the transition $t = (s, x, P, op, y, up, s')$ while x and y are parameterized input and output of the transition. Items of the *predicate_numbers* array and *function_numbers* array are identifiers of predicates and functions which guard the transition t.

Therefore, **structure *EFSM*** consists of the following data items.

```
struct EFSM
{
int s_number;
input *input_array;
int input_array_size;
output *output_array;
int output_array_size;
variable *variable_array;
int variable_array_size;
int *integers_array;
int integers_array_size;
function *functions_array;
int function_array_size;
predicate *predicates_array;
int predicates_array_size;
transition *transitions_array;
int transitions_array_size;
};
```

s_number is the number of states of the EFSM;.

input_array (of *input_array_*size items) and *output_array* (of *output_array_*size items) form the sets of parameterized inputs and outputs;

variable_array stores *variable_array_size* context variables.

If predicates or functions use constant integers then these integers are saved in the *integers_array*. The functions and the predicates are stored in the *functions_array* and the *predicates_array*. The set *T* of transitions is listed in the *transitions_array*.

IV. EXPERIMENTAL RESULTS

We experimented with several protocol EFSMs. The preliminary experimental results show that for several protocol EFSM the unfolding procedure at an appropriate abstraction level can be performed without limiting the maximal number of states of an equivalent FSM. Those protocols are POP3, SMTP, and TIME [6]. The reason is that the number of states of the corresponding protocol EFSMs (at an appropriate abstraction level) is up to four and the number of context variables is less than three while the number of transitions does not exceed 16. The equivalent FSM for POP3 EFSM has six states and 106 transitions while the equivalent FSM for SMTP EFSM has four states and 36 transitions. The TIME EFSM is rather small that is why the number of TIME FSM transitions is 12. More detailed information about performed experiments is presented in Table 1.

Pro-tocol	Num-ber of EFSM states	Number of EFSM context variables	Number of EFSM transi-tions	Number of equivalent FSM states	Number of FSM transi-tions
POP 3	4	2	16	6	106
SM TP	2	1	8	4	36
TIM E	2	0	2	2	12

Table 1. Preliminary experimental results

V. CONCLUSIONS

In this paper, we described the EFSM computer representation that is of a big help when automatically unfolding a given EFSM to an equivalent FSM. Such unfolding needs the explicit enumeration of all different configurations reachable from the initial EFSM state. As the enumeration can lead to the state explosion problem, the maximal number of an equivalent FSM is usually limited. We experimented with several protocol EFSMs and our preliminary experimental results show that the unfolding procedure (at an appropriate description level) can sometimes be performed without limiting the maximal number of states of an equivalent FSM. More experiments with different protocol EFSMs are needed in order to estimate the effectiveness of the developed software.

REFERENCES

[1] A. Faro and A. Petrenko. Sequence Generation from EFSMs for Protocol Testing. In Proc. of COMNET'90, Budapest, 1990.

[2] A. Petrenko, S. Boroday, R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 2004, 30(1), pp. 29-42.

[3] A. Petrenko and N. Yevtushenko. Testing from Partial Deterministic FSM Specifications. *IEEE Trans. on Computers*, 2005.

[4] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, N.Yevtushenko. Experimental evaluation of FSM-based testing methods. In: Proc. of the IEEE International Conference on Software Engineering and Formal Methods (SEFM05). Germany, pp. 23-32.

[5] V.A. Sibirjakova. Reverse Polish Notation: manual. Tomsk State University Publishers, 1997, 27 p.

[6] N.V. Spitsyna, A.V. Shabaldin. Web-programming: manual. Tomsk State University Publishers, 2002, 50 p.