ROMAI J., **2**, 1(2006),
175–180

ERROR HANDLING IN SOFTWARE SYSTEMS: MODELLING AND TESTING WITH FINITE STATE MACHINES

Radu Oprişa

Abstract Using the W method a test set of input sequences for the implementation of a software system starting from the Finite State Machine corresponding to specifications (SFSM) is obtained. To avoid halting, FSM corresponding to implementation (IFSM) must be completely specified. For the size reasons, usually specifications do not contain error-handling description for each unexpected input symbol. Due to this reason, for each state the designer must add in SFSM a corresponding error state and transitions to it for all unexpected input symbols. With this mechanism we are sure that IFSM is accepting all unexpected input sequences. But this will not prevent the problem of system recovery from error. In order to be able to perform a recovery from error we must allow the system to rich the previous state from each error state by adding for each error state a transition to the previous state. With this modeling approach we have the advantage of detecting all possible errors and the certitude of full error recovery in software systems at runtime.

Keywords: Finite State Machine, W method
2000MSC: 68N30

1. INTRODUCTION

Because the market necessities are growing, the software systems became more and more complex. Due to this, the effort to verify their correctness is also severely increased. The cheapest way to check if the implementation of a software system with respect to its specifications is an automatic testing is to be performed. For this purpose several methods had been developed, but all of them have specific limitations. An interesting approach is to model both system specifications and implementation with Finite State Machine and to determine a set of input sequences which can be used to prove or to invalidate their equivalence.

2. FUNDAMENTALS

Before generating the test set we need to introduce some basic notions about Finite State Machines. The following notation will be used. If A is a finite alphabet, denote by A^* the set of all finite sequences with members in A. Let ϵ denote the empty sequence. For $a, b \in A$ the concatenation

of the sequence a with b is denoted by $a \bullet b$. For $U, V \subseteq A^*$ we denote $U \bullet V = \{a \bullet b \mid a \in U, b \in V\}$.

Definition 2.1 *A Deterministic Finite State Machine (DFSM) is an association of five elements $M = (Q, Li, Lo, h, M_0)$ where:*

- *Q is a finite set of states;*
- *Li is a finite set of input symbols;*
- *Lo is a finite set of output symbols;*
- *$h : Q \times Li \to Q \times Lo$ is the behaviour (partial) function;*
- *M_0 is the initial state.*

Definition 2.2 *A DFSM M is said to be completely specified if h is a total function.*

Definition 2.3 *Let $M = (Q, Li, Lo, h, M_0)$ be a DFSM, $A \subseteq Li^*$ a set of input sequences and q, r two states in Q. Then q and r are called* $A-$equivalent *if both produce the same responses for each input sequence $a \in A$. Otherwise, they are said to be* $A-$ distinguishable*. If q and r give identical responses for any input sequence $a \in Li^*$ then q and r are said to be* equivalent*.*

Definition 2.4 *Let $M = (Q_M, Li, Lo, h_M, M_0)$ and $N = (Q_N, Li, Lo, h_N, N_0)$ be two DFSMs with the same sets of input and output symbols and $A \subseteq Li^*$. Then M and N are said to be* $A-$ equivalent *if their initial states M_0 and N_0 are* $A-$ equivalent*. Otherwise, M and N are said to be $A-$ distinguishable. If $A = Li^*$ then M and N are said to be* equivalent*.*

Definition 2.5 *A DFSM M is said to be* minimal *if any other equivalent DFSM has at least the same number of states as M.*

Definition 2.6 *A finite set $S \subseteq Li^*$ is called a* state cover *of a minimal DFSM $M = (Q, Li, Lo, h, M_0)$ if S contains the empty sequence and for every state $q \in Q$, other than M_0, S contains an input sequence a that may lead the machine from the initial state M_0 to q.*

In other words, a state cover is a set of input sequences that enables us to access any state in the machine from the initial state.

Definition 2.7 *Let $M = (Q, Li, Lo, h, M_0)$ be a minimal finite state machine. $T \subseteq Li^*$ is called a* transition cover *of M if $\forall q \in Q, \exists a \in T$ so that* a *leads the machine from the initial state M_0 to q and $\forall x \in Li, a \bullet \{x\} \in T$.*

In a DFSM $M = (Q, Li, Lo, h, M_0)$, for any state $q \in Q$ there are sequences in T that take M from M_0 in q and then attempt to exercise transitions with

all input symbols from Li whatever they exists or not. Remark that if S is a state cover of M, then $T = S \cup [S \bullet Li]$ is a transition cover of M.

Definition 2.8 *A finite set $W \subseteq Li^*$ is called* a characterisation set *of a DFSM $M = (Q, Li, Lo, h, M_0)$ if any two distinct states $q, r \in Q$ are $W-$ distinguishable.*

Definition 2.9 *Let $S = (Q_S, Li, Lo, h_S, S_0)$ and $I = (Q_I, Li, Lo, h_I, I_0)$ be two DFSMs having the same sets of input and output symbols. A finite set $Y \subseteq Li^*$ is called a test set for S and I if: S and I are $Y-$ equivalent $\Longrightarrow$ S and I are equivalent.*

The following well-known theorem is determining a test set which can be used to find out if two DFSM are equivalent or not.

Theorem 2.1 *Let $M = (Q_M, Li, Lo, h_M, M_0)$ and $N = (Q_N, Li, Lo, h_N, N_0)$ be two minimal DFSM, T and W a transition cover and* a *characterisation set for one of these machines respectively. Then they are isomorphic (behave identically) if M_0 and N_0 are $T \bullet Z-$ equivalent.*

In other words, if both DFSMs produce the same outputs when the sequences from $T \bullet Z$ are applied, then they are equivalent machines.

3. THE W METHOD

Let $S = (Q_S, Li, Lo, h_S, S_0)$ and $I = (Q_I, Li, Lo, h_I, I_0)$ be two DFSMs having the same sets of input and output symbols, where I has at least the same number of states as S. Let d be the difference between the number of states of I and S. If S and I are deterministic and completely specified and S is in addition minimal, then the W method determinates a test set which can be used to check if S and I are equivalent.

The test set is determined as follows $Y = S_c \bullet Li[d+1] \bullet W$, where $S_c \subseteq Li^*$ is a state cover of S, $W \subseteq Li^*$ is a characterisation set of S, $Li[k] = \{\epsilon\} \cup Li \cup \ldots \cup Li^k$ for $k \geq 0$.

4. ERROR STATES

Consider the DFSM corresponding to specifications of a software system (SFSM) from fig. 1. As we can see, this SFSM is not completely specified. This is a simple example, but especially for large systems, in practice frequently we have combinations of states and input symbols with no description of the behaviour in the specifications. If the implementation is performed with respect to this SFSM, then for all such combinations of states and input symbols, where the behaviour function is not defined, the program will halt. This

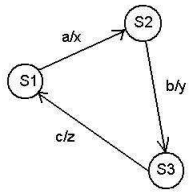

Fig. 1. Partially specified SFSM.

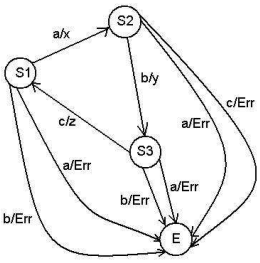

Fig. 2. SFSM with an error state.

is an undesirable behaviour for any software system especially when a large volume of user input data is required.

Example 4.1 *Consider the specifications of a software system which should allow user to fill input data in a form with several fields. At the end, the user should press "Submit" button in order to process data. At runtime the user fills the form, but by mistake press the "ALT+F4" key combination instead of "Submit" button. Because nothing is written in the specification about the behaviour when "ALT+F4" key combination occurs, the implementation has an uncontrolled behaviour.*

The most easy way for the designer to solve the problem of the uncontrolled behaviour in SFSM is to add a new state, an error state E, and also transitions to it for all possible states and input symbols which do not have corresponding transitions in the initial system. Fig. 2 presents the extended SFSM from fig. 1 with an error state and transitions to it. Now we have a controlled behaviour when an input symbol occurs for which we have no description in the specifications. But this is not enough for the user because the system will reach the E state and no possible action is defined from this state. Actually, even if the implementation will not halt, it is still impossible to recover input data from the system.

In order to solve this problem we need a more complex approach. Instead of defining only one error state, we will add an error state E_k for each state S_k of the initial SFSM and transitions T_{ke} for all unexpected input symbols from the SFSM state to the corresponding error state. In order to be able to perform also a system recovery from the error states we need to add supplementary

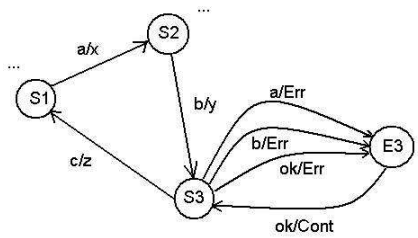

Fig. 3. SFSM with many error states.

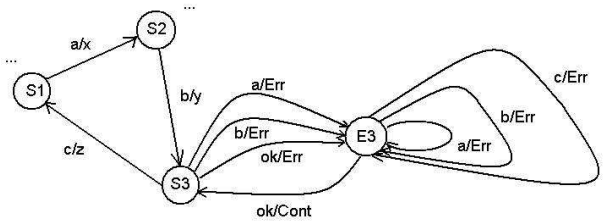

Fig. 4. SFSM with many error states.

transitions T_{kr} from error states to the previous state. The set of input symbols will be extended with a new one *ok* which will be used for all T_{kr} transitions with interpretation that user noticed the error state and allow system recovery. It is possible to avoid the use of a new input symbol and to reuse an existing one. Two output symbols will be added: Err which is used for all transitions which ends at E_k states, and $Cont$ which is used for T_{kr} transitions. Err is the system response when an error state is reached. To have only one response in case of any error in the system is not user friendly. Due to this fact, distinct output symbols can be added for each transition which ends at an error state. However, for test purposes one output symbol is enough.

The system from fig. 1 is extended as in fig. 3. To keep the figure simple only the error state for $S3$ is represented. This solves the problem of system recovery from error but the obtained SFSM still have a weak point. An error state is the initial state only for one transition which accepts only the input symbol *ok* (or the reused one). If another input symbol occurs when the system is in an error state, we are again in situation to obtain an uncontrolled behaviour. This can be solved by adding transitions for each error state and each unexpected input symbol to the error state itself. Such an extension is presented in fig. 4. To keep the figure simple, only the error state for $S3$ is represented.

5. CONCLUSIONS

Using this modelling technique and W testing method, the designer will be sure that the implementation will perform all requested operations described in the specifications. Moreover, all other actions initiated by the user

which are not described in the specifications will lead to an error state. Another big advantage is that the system can be recovered from any error state and this will eliminate the possibility of loosing data.

References

[1] Bălănescu, T., Gheorghe, M., Ipate, F., Holcombe, M., *Formal black box testing for partially specified deterministic finite state machines*, Fondations of Computing and Decision Sciences, **28**, 1 (2003), 17-28.

[2] Bălănescu, T. *Generalized stream X- Machines with Output Delimited Type*, Formal Aspects of Computing, **12** (2000), 473-484.

[3] Chow,T.S., *Testing software design modelled by finite state machines*, IEEE Trans. Softw. Engng, **4**, 3 (1978), 178-187.

[4] Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A., *Test selection based on finite state models*, IEEE Trans. Softw. Engng., **17**, 6 (1991), 591-603.

[5] Eilenberg, S., *Automata machines and languages*, Academic, New York, 1974.

[6] Gill, A., *Introduction to theory of finite state machines*, McGraw-Hill, New York, 1962.

[7] Hoare, C. A. R., *Communicating sequential processes*, Prentice Hall, Englewood Cliffs, 1985.

[8] Holcombe, M., *X-machines as a basis for dynamic system specification*, Software Engineering Journal, **3**, 2 (1988), 69-76.

[9] Holcombe, M., *An integrated methodology for the specification, verification and testing of systems*, Software testing, verification and reliability, **3** (1993), 149-163.

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

[11] Ipate, F., Holcombe, M., *Correct systems*, Springer, London, 1998.

[12] Ipate, F., Holcombe, M., *An integration testing method that is proved to find all faults*, Int. J. Comput. Math., **63**, 3/4 (1997), 159-178.

[13] Ipate, F., *Theory of X-machines with applications in specification and testing*, PhD thesis, Dept. of Computer Sci., Univ. of Sheffield, 1995.

[14] Ipate, F. and Holcombe, M., *Another look at computation*, Informatica, **20** (1996), 359-372.

[15] Ipate, F., Holcombe, M., *A method for refining and testing generalized machine specifications*, to appear in Int. Jour. Comp. Math. 1998. [WWW-http://www.dcs.shef.ac.uk/ wmlh/]

[16] Ipate, F., Holcombe, M., *A theory of refinement and testing for X-machines*, submitted.

[17] Ipate, F., Bălănescu, T., *Refinement in finite state machine testing*, IOS Press, 2004.

[18] Morgan, C., *Programming from specifications*, Prentice Hall, Englewood Cliffs, 1994.

[19] Selec, B., Gullekson, G., Ward, P., *Real-time object-oriented modeling*, Wiley, New York, 1994.

[20] Sommerville, I., Sawyer, P., *Requirements engineering - a good practice guide*, Wiley, New York, 1997.