Scientia Iranica, Vol. 8, No. 4, pp 241–249

SCIENTIA
IRANICA

Testing Reactive Systems on the Basis of Formal Specifications

S. Sadeghipour[1]

Using Extended Finite State Machines (EFSMs) in the specifications of reactive systems increases the preciseness and understandability of requirements. This paper focuses on the problem of testing based on EFSMs. While testing based on Finite State Machines (FSMs) has been extensively investigated in the literature, issues concerning testing based on EFSMs have not been adequately addressed. Here, after providing an overview of testing methods based on FSMs, a novel strategy for testing based on EFSMs is presented.

INTRODUCTION

Today software systems play a significant role in the daily life and the users of such systems make high demands on their quality. A major aspect of the software quality is the correctness of software with respect to its functional requirements. Testing is the primary method through which the producer of software and the user or customer gain confidence that these requirements are fulfilled. On the other hand, the use of formal methods in the development of software promises high product quality, because the precision of specifications written in a formal language forces specifiers to express requirements and design decisions in a clear and unambiguous manner. Formal methods use mathematical notations and proofs in the specification, analysis and design of software systems. The core of a formal method is a specification notation, which has to possess a well-defined syntax and semantics. Some examples for formal specification notations are Z, VDM, SDL, CSP and LOTOS.

Although the well-structured use of formal methods contributes to avoiding errors during the construction process of software and provides the prerequisites for using tool-supported verification methods, current proof techniques and tools are mostly too expensive to be efficiently used for complex tasks such as the verification of an implementation against its specification. Therefore, testing is the main verification method for software developers who use formal techniques. Formal methods can improve the quality and efficiency of test activities. The formal specification of the test object could be a solid basis for a systematic and tool-supported test case and test trace derivation, test input data selection and test evaluation leading to a cost-effective software development.

In this paper, testing reactive software systems is considered. In contrast to a transformational system which accepts an input, produces the corresponding output and terminates, reactive systems are characterized by their continuous reaction to various inputs from their environment. Examples include operating systems, communication protocols, man-machine interfaces of word processor, data base programs and the software components used in technical systems (embedded software). Finite State Machines (FSMs) build a simple and easily understandable formalism and suggest themselves to formally specific reactive systems. However, real systems usually process a large amount of data and, therefore, Extended Finite State Machines (EFSMs), which are FSMs extended by data variables, are frequently used to specify reactive systems. The main contribution of this paper to the methodology of software testing is a strategy for testing based on EFSMs. In order to make the reader familiar with the relevant research background, a brief description of testing methods based on FSMs is also provided. The next section explains the terminology of testing used in this paper.

TESTING TERMINOLOGY

Concerning the test basis, there is a dichotomy in the software test: Tests based on program specification are called black-box tests and tests based on program structure are called white-box tests. Testing programs against their specifications is decisive for detecting the

1. *DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, D-10559 Berlin, Germany.*

missing functionalities, i.e., requirements which are not implemented. Therefore, when testing a piece of software, the emphasis should be laid on specification based testing. However, to realize an effective test strategy, the black-box test should be accompanied by an appropriate white-box test in order to execute program branches not covered by test cases derived from the specification. This paper focuses on specification based testing.

In testing reactive systems, one can distinguish between function and trace tests. Function test refers to the test of individual functions of the software which are responsible for the transformation of data and can be triggered by certain inputs in certain internal states. Trace test, or testing the dynamic behavior, means the test of traces of functions. For the function test, at first, test cases are derived from the specification. A test case describes a certain input situation. Then, test input data are selected for the derived test cases. After the test execution, i.e., executing the test object with the selected input data, the test evaluation checks whether each pair consisting of the test input data and the corresponding output produced by the test object satisfies the specification. For the test of the dynamic behavior of a system, the generation of test traces builds the first activity. Test traces are sequences of functions. The other activities are similar to the function test, i.e., test input data selection, test execution and evaluation, while here test input traces have to be selected as input data in order to trigger the functions contained in a trace.

TESTING BASED ON FINITE STATE MACHINES

First, two basic definitions are provided and then, the test methods based on FSMs are described.

Definition 1

A Finite Automaton (FA), also called an acceptor, is a 4-tuple (S, Σ, q_0, δ), where S is a finite set of states, Σ is a finite set of symbols, called the automaton alphabet, $q_0 \in S$ is the initial state, and $\delta : S \times \Sigma \leftrightarrow S$ the transfer relation. A finite automaton can be visualized by a directed graph.

Definition 2

A Finite State Machine (FSM), also called a Mealy automaton, is a 5-tuple $M = (S, \Sigma, \Delta, q_0, f)$, where S is a finite set of states, Σ and Δ are finite sets of input and output symbols, respectively, $q_0 \in S$ is the initial state and $f : S \times \Sigma \leftrightarrow S \times \Delta$ the transfer and output relation. In a deterministic FSM, f is a partial function. For example, $f(q_j, i) = (q_k, o)$ means that if an input $i \in \Sigma$ is applied to M in state q_j, a transition t_{jk} is executed and M moves to state q_k and produces output o. Such a transition is denoted by $t_{jk} = (q_j, q_k, i/o)$. An FSM is considered to be completely specified if in each state and for each input symbol, there is a transition leaving that state.

For the testing methods based on FSMs, the implementation to be studied has to be specified as a deterministic FSM M_s. (This paper focuses on deterministic specifications. In the research field testing based on Labeled Transition Systems (LTSs), and testing methods based on non-deterministic machines are investigated, e.g., [1,2].) Moreover, it is assumed that the behavior of the implementation can also be modelled by a deterministic FSM M_i having the same input and output alphabet as M_s. The aim of testing is to check the equivalence of M_i and M_s, i.e., whether they accept the same traces of inputs and generate the same traces of outputs for each accepted input. In the most general case, M_i may differ from M_s in:

1. The number of states (i.e., M_i may have missing or extra states),
2. The number of transitions (i.e., M_i may have missing or extra transitions),
3. The implementation of the output and transfer functions (i.e., some transitions of M_i may produce erroneous outputs or enter erroneous states) [3].

In general, extra states or transitions in M_i cannot be detected with a finite number of input traces. Hence, a test hypothesis which is often used is that the number of states of the formal model of the implementation is equal to the number of states of the specification. The FSM specification is required to be minimal and completely specified. The minimality is needed in order to exclude equivalent states, which result in confusion in the identification of the state reached by the implementation (see the third step of the correctness regarding checking below). The completeness of the specification is required, because otherwise the implementation can have an arbitrary behavior for the input traces outside the specification domain. Moreover, each state has to be reachable, so that the testing procedure is able to check each state by the application of an input trace.

As an example, a specification machine M_s is considered in Figure 1 and the pseudocode of an erroneous implementation M_i is presented in Figure 2. The formal model of the implementation is also shown in Figure 2. The specification is deterministic, minimal and completely specified. It is characterized by the set of states $S = \{P, Q, R\}$, the input alphabet $\Sigma = \{0, 1\}$ and the output alphabet $\Delta = \{a, b\}$. The transfer and output function f of M_s is described by the directed graph in Figure 1. The automaton model of

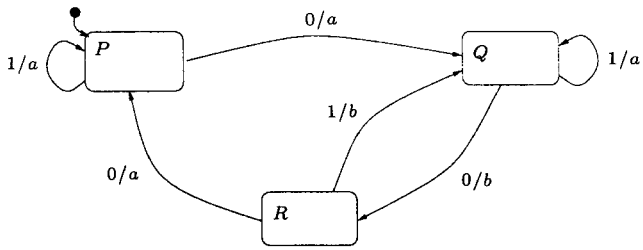

Figure 1. Specification of finite state machine M_s.

the implementation in Figure 2 shows that the implementation differs from the specification in a missing transition, faulty implemented transfer function and output function. Assuming that the formal model of implementation does not have more states than the specification, testing the equivalence between the specification and implementation is reduced to testing whether or not each transition of the FSM specification has been implemented correctly. The procedure of checking the correctness of the implementation of a transition $t_{jk} = (q_j, q_k, i/o)$ consists of three steps [3-5]:

1. M_i is brought (e.g., from the initial state q_0) to state q_j by an input trace u,
2. An input i is applied to M_i and the output produced by M_i is checked to see whether it is o,
3. The state reached by M_i after the application of input i is checked by a characterizing input trace w to see whether it is q_k.

The input trace uiw is called a test input trace for the transition t_{jk}. Different methods of test input trace generation, e.g., Distinguishing Sequence (DS), Unique Input/Output (UIO) and W-Method differ in the third step mentioned above, i.e., in the way the identification of the state reached after the transition to be tested has been executed [3,6]. A distinguishing sequence is an input trace that produces a different output for each state. A unique input/output for state s_i is a trace of input/output pairs that has the property that the output produced at s_i for this trace differs from that produced from any other state $s_j \neq s_i$ [7]. For an arbitrary FSM, the existence of distinguishing sequences or a unique input/output is not guaranteed. In comparison, W-Method which suggests a characterization set of input traces to distinguish between the states of an FSM is applicable in all cases. Set of input traces W is a characterization set of an FSM M if for each pair of states q and q', there is at least one input trace in W for which the machine produces different outputs at q and q'. Thus, given a transition to be tested, that transition must be executed together with all the members of the characterization set. Consequently, the testing effort according to W-Method is generally more than the other two methods. Moreover, the W-Method does not require the number of states of the specification and the formal model of the implementation to be equal, but only the maximum number of the implementation states to be known. W-Method is described below in detail.

W-Method

W-Method, or the method of characterization sets [4], has been proven to be capable of showing the equivalence between the specification and the formal model of the implementation. In order to generate test input traces for an FSM according to W-Method, one should build a transition cover of the machine. A transition cover of an FSM is a set of input traces that for each input trace reaches the source state of a certain transition and covers it [6]. A transition cover can be built by deriving a reachability tree, also called a testing tree, from the FSM. If the number of states of the formal model of the implementation is assumed to be equal to the number of states of the specification, the set of test input traces T is built by the concatenation of the transition cover P and characterization set W of M_s: $T \widehat{=} P \circ W$. (If the implementation automaton is assumed to have more states than the specification automaton, then T is computed in a more complex

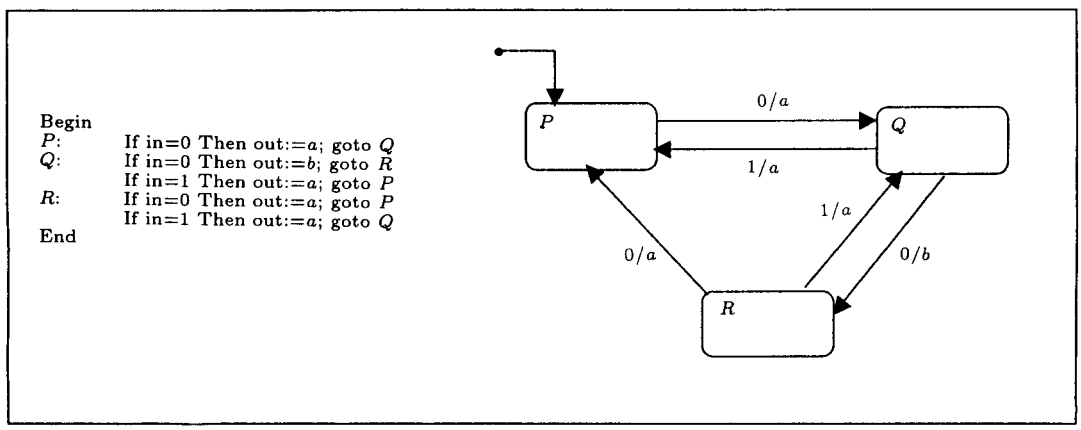

Figure 2. Pseudocode implementation and its finite state machine M_i.

Table 1. Test table of M_i against M_s.

Test Input Trace	Expected Output	Real Output
$\langle 10 \rangle$	$\langle aa \rangle$	$undefined$
$\langle 010 \rangle$	$\langle aab \rangle$	$\langle aaa \rangle$
$\langle 110 \rangle$	$\langle aaa \rangle$	$undefined$
$\langle 0010 \rangle$	$\langle abbb \rangle$	$\langle abab \rangle$
$\langle 0110 \rangle$	$\langle aaab \rangle$	$\langle aa(undefined) \rangle$
$\langle 00010 \rangle$	$\langle abaaa \rangle$	$\langle aba(undefined) \rangle$
$\langle 00110 \rangle$	$\langle abbab \rangle$	$\langle abaaa \rangle$

way [4].) An algorithm for building characterization sets has been given in [8] (Algorithm 4.1, p 92).

As an example, the specification machine M_s of Figure 1 is considered. The transition cover P of M_s is:

$$P = \{\epsilon, \langle 0 \rangle, \langle 1 \rangle, \langle 00 \rangle, \langle 01 \rangle, \langle 000 \rangle, \langle 001 \rangle\},$$

where ϵ is the empty trace. Set W consisting only of the input trace $\langle 10 \rangle$ can be considered to be the characterization set of M_s, because it produces a different output at every state of M_s. Assume that the formal model of an implementation, regarded as the test object, has the same number of states as the specification, namely three. Then, the set of test input traces T is built as follows:

$$T \widehat{=} P \circ W = \{\langle 10 \rangle, \langle 010 \rangle, \langle 110 \rangle, \langle 0010 \rangle, \langle 0110 \rangle,$$
$$\langle 00010 \rangle, \langle 00110 \rangle\}.$$

Table 1 shows the expected outputs and the corresponding real outputs computed by the faulty implementation M_i, illustrated in Figure 2, for the test input traces from T.

TESTING BASED ON EXTENDED FINITE STATE MACHINES

It is known that FSMs and FAs are merely capable of computing regular expressions, i.e., their computation power is much less than the general Turing Machines. Therefore, FSMs and FAs can hardly be used for the specification of reactive systems. A solution to this problem is to extend finite automata by data variables leading to Extended Finite State Machines (EFSMs). In this manner, the expression power of the finite machine formalism is notably increased while its understandability is maintained. EFSMs are finite automata whose transitions describe transformations on (possibly infinite) data types. The data space of such a machine is characterized by sets of input, output and internal variables. The details concerning data transformations are hidden in transition labels and are specified separately within a certain formal syntax. The formal notation used in this paper is a combination of Z [9] and finite automata. This notation and a strategy for testing based on EFSMs are explained in the consequent sections.

Specification Notation

The specification notation (and the test strategy presented in the next section) is described with the aid of a simplified version of the specification of a cruise control system for cars [10] serving as the running example throughout the rest of the paper. In this section, the specification of the cruise control and, at the same time, an explanation for the main components of the notation used are presented. As mentioned previously, the specification notation is a combination of Z and finite automata. (The Z part of the notation is enriched by some concepts of the object oriented version of Z, called Object-Z). Z is a model-oriented specification language based on the first order predicate logic and set theory. The central concept of Z is a schema, which is used to describe the operations and data state of the software to be specified. A schema generally consists of a signature, which declares the variables and their types and a predicate, which expresses the properties to be fulfilled. In addition to schemas, Z possesses syntactical constructions for specifying data types, constants and functions.

The cruise control system, *CruiseControl*, supports car drivers by controlling the vehicle speed. In the simplified version presented here, this is controlled by the cruise control lever having three positions 'on', 'off' and 'middle'. The driver can define the current speed as the requested speed by setting the control lever on 'on'. By applying the brake or setting the control lever on 'off', the *CruiseControl* is deactivated. The lever is set automatically to the 'middle' position after each activation. Figure 3 shows an automaton specifying the dynamic behavior of the *CruiseControl* which has two states, *passive* and *active*, characterizing the control modes, and two transitions responsible for operations described above.

Figure 4 illustrates the data types and the operations of *CruiseControl* specified in Z. The Z schema defined by the key word D_{ATA} describes the private data space of the specification. The predicate of a data schema is an invariant that must be valid

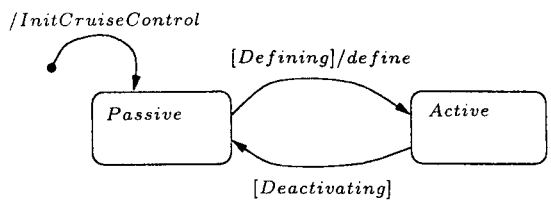

Figure 3. Dynamic behavior of *CruiseControl*.

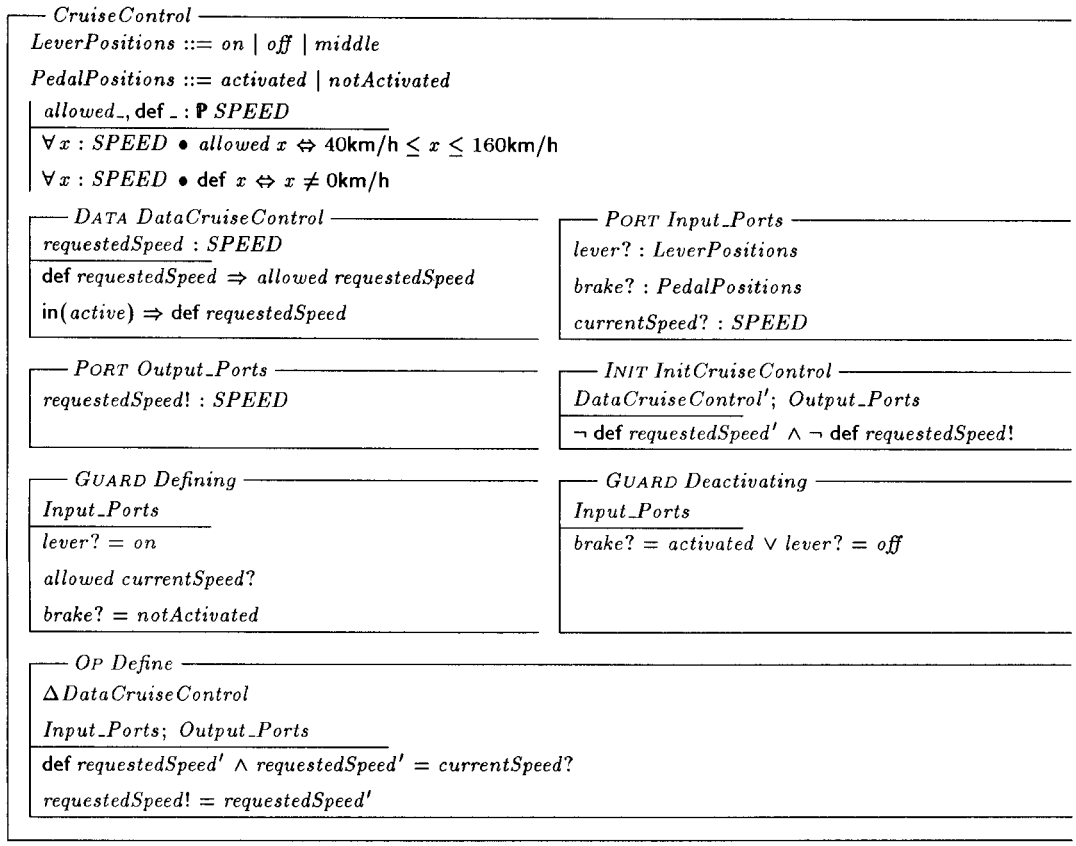

Figure 4. Specification of *CruiseControl*.

before and after the execution of each operation. The variable *requestedSpeed* declared in the schema *DataCruiseControl* models the speed requested by the driver and is undefined after the system initialization. In the full version of *CruiseControl*, this variable is used to set the requested speed after resuming the cruise control system. The data type *SPEED* is from the μSZ library for dimensioned physical quantities. Using this library makes it possible to apply physical units (e.g., km/h for speed) to numbers. The predicate **in** is a built-in predicate of μSZ. For example, **in** (active), as used in the schema *DataCruiseControl*, expresses that the statechart of the specification resides in the state active.

In the port schemas, variables for the communication of a software component with its environment and other components are declared. According to Z conventions, incoming variables are decorated with a question mark "?" and outgoing ones with an exclamation mark "!". The initialization schema defines the initial values of the internal data and the output variables. It is attached to the transition entering the initial state of the automaton. Variables decorated by a prime "′" denote the value of the corresponding undecorated variable after the execution of the operation. An automaton transition is labelled by a pair $[guard]/op$, called a transition function or a transition relation, where *guard* and *op* are specified by individual Z schemas. A guard describes the condition for a transition to be triggered. An operation specifies the data transformations to be carried out by the transition. The term ΔS in the signature of an operation schema *Op* means the inclusion of the variables of the S in *Op* in their simple and primed form. The transition label $[guard]/op$ is logically interpreted as $guard \wedge op$. The absence of any operation (such as label [*Deactivating*] in Figure 3) means that the operation does not change the internal data variables and produces no output. In order to derive test traces, it is crucial to have specifications with their whole information made explicit. Therefore, an operation *NoAction* is defined and attached to the corresponding transition:

$$O_p\ NoAction \widehat{=} [\Delta\, DataCruiseControl; Output\_Ports \mid \neg \mathrm{def}\, requestedSpeed! \wedge requestedSpeed' = requestedSpeed].$$

Moreover, to specify the behavior of an automaton for the possible inputs which are not covered by the guards leaving a state, a loop-back transition is attached to that state. Figure 5 shows the complete automaton

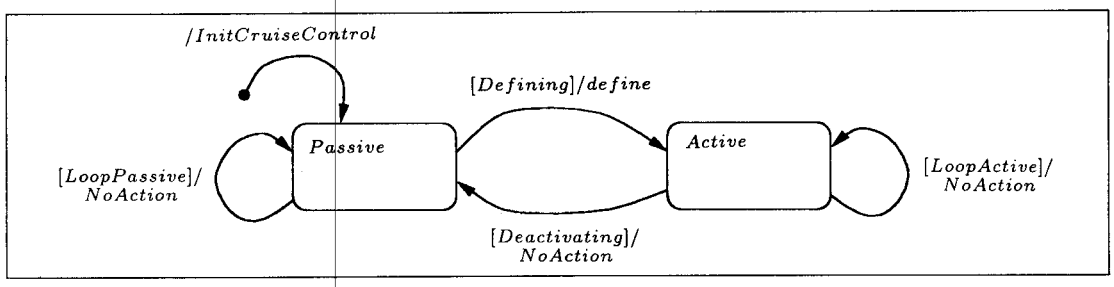

Figure 5. Complete statechart of *CruiseControl*.

of *CruiseControl*, having the loop-back transitions and *NoAction* operations made explicit, with:

$$LoopActive \widehat{=} \neg Deactivating \text{ and } LoopPassive$$

$$\widehat{=} \neg Defining.$$

Separate Function and Trace Test

The starting point for designing a test strategy based on EFSMs is the well-known test trace generation approach based on FSMs, described previously. The problem of carrying over the test methods based on FSMs to EFSMs is twofold. On one hand, a transition in an FSM accepts an input and produces an output. Once it has been tested successfully, it can be regarded as correctly implemented. However, in an EFSM, the transitions transform a set of variable values. The successful test of a transition for some variable values does not guarantee that the transition is correctly implemented for other values of variables. The other problem concerns the recognition of an executed transition. In FSMs, different outputs are distinguishable, yet in EFSMs, different operations may produce the same output. This analysis leads to the strategy of separate function and trace test (SFTT strategy) which is motivated by the test method based on X-machines [11] and uses the test trace generation methods based on FSMs. The SFTT strategy can be applied to an EFSM if the specification and the development process fulfil the following conditions:

- The transitions of the EFSM have to be implemented as individual testable units. If so, then they can be tested separately and their correctness can be assumed during the trace test. Therefore, the first problem mentioned above does not arise, because the correctness of the behavior of each transition in different situations is not subject to trace test;
- Each operation produces a distinguishing output at least for a certain input (output distinguishability). This condition solves the second problem mentioned above, because it ensures that each transition can be identified through its outputs;
- The guard of each operation is satisfied for the input producing the distinguishing output mentioned above, independent of the values of the internal variables (feasibility). This condition guarantees that all test traces derived from the machine are actually feasible.

In order to avoid the complex analyses needed for checking the conditions of output distinguishability and feasibility, all the transitions of an EFSM are augmented with special inputs and outputs. They are added to the transitions regardless of whether or not they possess any of the desired conditions and are used only for testing. In an EFSM augmented in this manner, each transition can be triggered by its special testing input, regardless of the values of the internal variables, and each transition reports its execution through the special testing output. In the following, this kind of augmentation for the specification *CruiseControl* is shown. The automaton of this specification has four differently labelled transitions (Figure 5). Therefore, two enumeration types containing four values for the testing input and the testing output variables are introduced:

$$TestingInputs ::= defininingIn | deactivatingIn$$

$$| loopPassiveIn | loopActiveIn$$

$$TestingOutputs ::= defineOut | deactivateOut$$

$$| opPassiveOut | opActiveOut$$

The port schemas *Input_Ports* and *Output_Ports* are extended by the input variable *testingInput*? from the type *TestingInputs* and the output variable *testingOutput*! from the type *TestingOutputs*, respectively. Each guard checks whether the corresponding testing input variable is set. For example, the guard *Defining* is extended as follows:

Guard Defining *Input_Ports*
$(lever? = on \wedge allowed\ currentSpeed? \wedge brake?$ $= notActivated) \vee$ $(testingInput? = defininingIn)$

Since the transitions labelled by *NoAction* operation in Figure 5 have different guards, they are considered

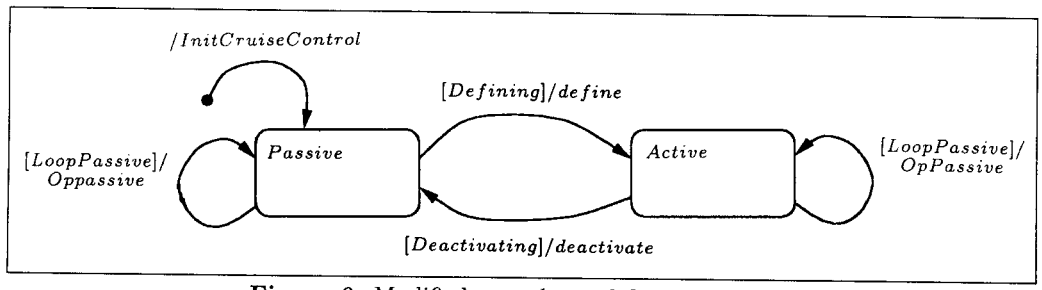

Figure 6. Modified statechart of *CruiseControl*.

to be different from each other and have to produce different values of the testing output variable. Therefore, three new operations, *OpPassive*, *OpActive* and *Deactivate*, are introduced and the different *NoAction* operations are replaced by them. Figure 6 shows the modified automaton of *CruiseControl*. As an example of the augmented operations, the Z schema of *OpPassive* is provided:

$$
\begin{array}{|l}
\hline
O_p\ OpPassive \\
\Delta\ DataCruiseControl \\
Input\_Ports; Output\_Ports \\
\hline
\neg\ \text{def}\ requestedSpeed! \wedge requestedSpeed' \\
= requestedSpeed \\
testingInput? = loopPassiveIn \\
\Rightarrow testingOutput! = opPassiveOut \\
\hline
\end{array}
$$

The original predicate of the operation is conjugated with a predicate expressing the presence of the distinguishing testing output, provided that the corresponding testing input has been presented.

As the name of the SFTT strategy reveals, the function and the trace test proceed separately. At the first stage, the system functionalities, which are represented by automaton transitions, are tested, and at the next stage, while the correctness of individual functions is assumed, the trace test is designed and executed. In the following subsections, the activities related to the trace test are described.

Test Trace Derivation

For deriving test traces from an EFSM which has been augmented according to the procedure described previously, the test trace generation methods based *on FSMs can be used.* W-method is applied to the automaton of *CruiseControl*. For this purpose, the transition labels of the automaton are replaced by simple symbols. This leads to a finite automaton, as shown in Figure 7, called the abstract FA of the original EFSM. The transition cover of the abstract FA of *CruiseControl* is given by:

$$P = \{\epsilon, \langle a\rangle, \langle c\rangle, \langle b\rangle, \langle d\rangle, \langle cb\rangle, \langle cd\rangle, \langle ca\rangle, \langle cc\rangle\}.$$

The set $W = \{a\}$ is a characterization set, because a is accepted at the *passive* state but not at the

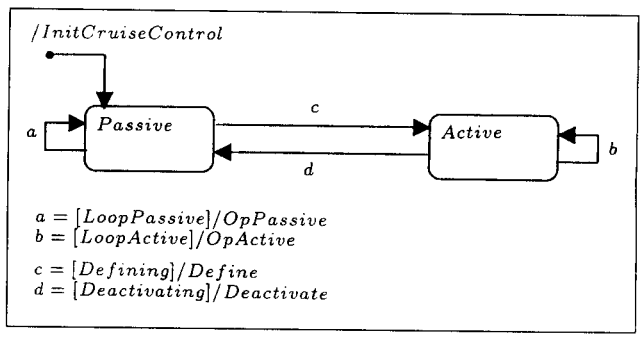

Figure 7. Abstract FA of the EFSM of *CruiseControl*.

active state. Assuming that the formal model of the implementation also has two states, the set of test traces is:

$$T = P \circ W = \{\langle a\rangle, \langle aa\rangle, \langle ca\rangle, \langle ba\rangle,$$
$$\langle da\rangle, \langle cba\rangle, \langle cda\rangle, \langle caa\rangle, \langle cca\rangle\}.$$

A trace which is a prefix of another one can be eliminated because it is tested by a longer trace. Thus, the final set of test traces is:

$$T' = \{\langle aa\rangle, \langle ba\rangle, \langle da\rangle, \langle cba\rangle, \langle cda\rangle, \langle caa\rangle, \langle cca\rangle\}.$$

Test Input Data Selection

The input data for a test trace is a trace of the values of the input variables capable of triggering the operations of the test trace. The selection of the test input data proceeds as follows:

- For every test trace $t = \langle f_1 \ldots f_n\rangle$ accepted by the abstract FA of the considered EFSM, a trace $\langle in_1 \ldots in_n\rangle$ of the values of the variable *testingInput*? corresponding to the transitions appearing in t is selected. The values of all other input variables are arbitrary.
- If test trace $t = \langle f_1 \ldots f_n\rangle$ is not accepted by the abstract FA, then there is an initial segment of t, $t' = \langle f_1 \ldots f_k\rangle$, $k < n$, which is accepted by the automaton (for $k = 0$, t' is the empty trace). The trace of the values of the variable *testingInput*? corresponding to t, $\langle in_1 \ldots in_k in_{k+1}\rangle$ is, then, composed of the input trace $\langle in_1 \ldots in_k\rangle$ corresponding to t', built as described above, and an input in_{k+1} which

Table 2. Test input data and the expected outputs for traces of T'.

Test Trace	Values of Testing Input?	Values of Testing Output!
$\langle aa \rangle$	$\langle loopPassiveIn\ loopPassiveIn \rangle$	$\langle opPassiveOut\ opPassiveOut \rangle$
$\langle ba \rangle$	$\langle loopActiveIn \rangle$	$\langle (not\ opActiveOut) \rangle$
$\langle da \rangle$	$\langle deactivatingIn \rangle$	$\langle (not\ deactivateOut) \rangle$
$\langle cba \rangle$	$\langle definingIn\ loopActiveIn\ loopPassiveIn \rangle$	$\langle defineOut\ opActiveOut\ (not\ opPassiveOut) \rangle$
$\langle cda \rangle$	$\langle definingIn\ deactivatingIn\ loopPassiveIn \rangle$	$\langle defineOut\ deactivateOut\ loopPassiveOut \rangle$
$\langle caa \rangle$	$\langle definingIn\ loopPassiveIn \rangle$	$\langle defineOut\ (not\ opPassiveOut) \rangle$
$\langle cca \rangle$	$\langle definingIn\ definingIn \rangle$	$\langle defineOut\ (not\ defineOut) \rangle$

would have caused the specification to exercise the function f_{k+1} if it had been accepted by the abstract FA.

Table 2 provides the traces of the values of the variable *testingInput*? for traces from T'.

Test Evaluation

Due to the distinguishability of testing outputs for each transition, the procedure of test evaluation is straightforward. A test is successful if the expected trace of testing outputs is observed.

In order to explain the identification of expected outputs, a distinction is made between the test traces which are accepted by the abstract FA and those which are not accepted.

- Test trace t is accepted by the abstract FA of the specification. In this case, for each element of the test trace, which is the name of a transition function, its specific value of the testing input variable has been applied to the implementation. Therefore, for each value of the testing input variable, an observer expects its corresponding output. For example, consider the trace $\langle cda \rangle$ from Table 2, which is accepted by the abstract FA of Figure 7. According to the second column of the table, the applied test input trace is $\langle definingIn\ deactivatingIn\ loopPassiveIn \rangle$. According to the kind of augmentation of the specification described before, this input trace must generate the output trace $\langle defineOut\ deactivateOut\ loopPassiveOut \rangle$;
- According to the construction of the test input data, described in the previous subsection, the expected output for a test trace not accepted by the abstract FA differs from the above case only in the last member. In a correct implementation, the last value of the testing output variable for such traces may be undefined or be any value except the value corresponding to the last testing input. As an example, see the last row of Table 2 for the test trace $\langle cca \rangle$, where the ouput $defineOut$ is not allowed to occur twice.

The last column of Table 2 shows the expected traces of the values of the testing output variable *testingOutput*! for the test traces from T'.

CONCLUSIONS

An overview of testing methods based on finite state machines and the strategy of Separate Function and Trace Test (SFTT) on the basis of extended finite state machines are presented. For the application of SFTT strategy to an EFSM, the specification and development process have to fulfil some conditions. The key condition is the presence of individual testable units for the transitions of the EFSM. For the cases considered, this condition does not hold, therefore, an alternative strategy, called Combined Function and Trace Test (CFTT), described in [12] was developed. In order to facilitate the usage of these test strategies in the industrial practice of software development, a tool environment is being developed at the Software Technology Labour of DaimlerChrysler.

In addition to testing, model checking is a verification method based on finite state machines, which has become popular in recent years [13,14]. The objective of model checking is to check the correctness of a model against a property. The property might be taken from the requirements specification, which is usually written in a temporal logic language. The checking is accomplished by the complete exploration of the state space of the model. The objective of a black-box testing strategy, as described in this paper, is basically different from the objective of model checking: Model checking applies a white-box approach and investigates the relation between a model, having a known structure, and a specification. However, testing tries to check the correctness of a program, considered as a black-box, against a specification (or a model). In practice, model checking should be done after building

a model in order to detect the modelling errors. Testing is accomplished after the implementation in order to detect programming errors.

ACKNOWLEDGMENTS

The author wishes to thank his colleagues Professors Harbhajan Singh and Kirill Bogdanov for their cooperation and many fruitful discussions on testing based on formal specifications.

REFERENCES

1. Brinksma, E. "A theory for derivation of tests. In P.H.J.", *The Formal Description Technique LOTOS*, van Ejik, C.A. Vissers and M. Diaz, Eds., North-Holland (1989).
2. Peleska, J. and Siegel, M. "From testing theory to test driver implementation", *FME'96, Industrial Benefit and Advances in Formal Methods*, M.C. Gaudel and J. Wood-cock, Eds., pp 538-556, Springer (1996).
3. Ural, H. "Formal methods for test sequence generation", *Computer Communications*, **15**(5), pp 311-325 (1992).
4. Chow, T.S. "Testing software design modeled by finite-state machines", *IEEE Transactions on Software Engineering*, **SE-4**(3), pp 178-187 (1978).
5. Lee, D. and Yannakakis, M. "Principles and methods of testing finite state machines - A survey", *Proc. of the IEEE*, **84**(8), pp 1089-1123 (Aug. 1996).
6. Fujiwara, S., Bochman, G.V., Khendek, F., Amalou, M. and Ghedamsi, A. "Test selection based on finite state models", *IEEE Transactions on Software Engineering*, **17**(6), pp 591-603 (June 1991).
7. Hierons, R.M. "Extending test sequence overlap by invertibility", *The Computer Journal*, **39**(4), pp 325-330 (1996).
8. Gill, A., *Introduction to the Theory of Finite-State Machines*, McGrow-Hill (1962).
9. Spivey, J.M., *The Z Notation. A Reference Manual*, Prentice Hall, New York, 2nd Ed. (1992).
10. Grieskamp, W., Heisel, M. and Dörr, H. "Specifying embedded systems with statecharts and Z: An agenda for cyclic software components", In *Proc. Formal Aspects of Software Engineering - FASE'98*, number 1382 in LNCS Springer (1998).
11. Ipate, F. and Holcombe, M. "An integration testing method that is proved to find all faults", *International Journal on Computer Mathematics*, **63**, pp 150-178 (1997).
12. Sadeghipour, S. "Testing cyclic software components of reactive systems on the basis of formal specifications", Ph.D Thesis, Technische Universität Berlin, Fachbereich Informatik (1998).
13. Burch, J.R., Clarke, E.M. and McMillan, K.L. "Symbolic model checking: 10^{20} states and beyond", *Information and Computation*, **98**, pp 142-170 (1992).
14. Grumberg, O. ad Long, D.E. "Model checking and modular verification", *ACM Transactions on Programming Languages and Systems*, **16**(3), pp 843-871 (May 1994).