© The Author 2008. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
Advance Access publication on May 20, 2008
doi:10.1093/comjnl/bxn027

Test Selection for Hierarchical and Communicating Finite State Machines

FLORENTIN IPATE

Department of Computer Science, University of Pitesti, Str Targu din Vale 1, 0300 Pitesti, Romania
Corresponding author: florentin.ipate@ifsoft.ro

State-based languages are widely used for modelling systems that have an internal state, such as communications protocols and embedded control systems. As testing is a vital part of system development, this has led to much interest in testing from *finite state machines* (*FSMs*). However, complex systems are seldom designed in one step; usually, the design is constructed gradually, through a process of *refinement*. In the case of state-based models, this may lead to a hierarchy of machines. Furthermore, some of the components of the hierarchy may exhibit concurrent behaviour. In this paper, we present a method for generating tests for a hierarchical FSM by reusing and refining the tests for the FSM components of the hierarchy. The method is also adapted for testing a system of communicating FSMs, in which the communication is one-directional, from one master to one or more slaves.

Keywords: finite state machines; testing; specification-based testing

Received 7 June 2007; revised 8 February 2008

1. INTRODUCTION

State-based languages, such as Statecharts [1] and SDL [2] are widely used for modelling systems that have an internal state, such as communications protocols and embedded control systems. State diagrams have also become a standard model for representing object behaviour; in particular, UML [3] adopted this model. As testing is a vital part of system development, this has led to much interest in testing from *finite state machines* (*FSMs*) [4–9]. Given an FSM specification, whose transition diagram is known, and an implementation, which is a 'black box' for which we can only observe its input/output behaviour, we want to select a set of test sequences that check whether the implementation under test conforms to the specification. This is called *conformance testing*. Many test-selection methods rely on the assumption that the implementation cannot have more states than the specification; among these, the best known are based on Transition Tours [7], (Multiple) Unique Input Output (UIO) sequences [7, 8], Characterizing Sequences [5] and Distinguishing Sequences [7, 9] (the methods are enumerated in increasing order of their fault-detection capability [6]). Although this assumption is quite restrictive, these methods have been successfully used in testing of network protocols [6–8]. A less restrictive assumption is required by the *W*-method [10]: this generates test suites that guarantee the correctness of the implementation provided that the number of states of the implementation remains below a known upper bound.

Complex systems are seldom designed in one step. Usually, the design is constructed gradually, through a process of refinement. In the case of state-based models, this may lead to a hierarchy of machines [1, 11]. Furthermore, some of the components of the hierarchy may exhibit concurrent behaviour. In conformance testing, hierarchical and concurrent models are usually turned into behaviourally equivalent FSMs, which are then used as the basis for test generation [12, 13]. However, this approach suffers from the state explosion problem [4] and, furthermore, the refinement used in the design process is not reflected in the construction of the implementation and of the test data. An alternative approach is to develop the implementation in parallel with the design, that is each change in the design is accompanied by an appropriate change in the implementation. In this case, tests can also be constructed and refined in parallel with the specification, and so test generation will be kept consistent with the design and we will avoid leaving testing to the final pre-release stage. Furthermore, by developing the implementation in parallel with the design, some implementation faults are avoided from the outset and so fewer test cases will be needed.

In this paper, we present a method for generating tests for a hierarchical FSM by reusing and refining the tests for the FSM

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

components of the hierarchy. Test generation from hierarchical FSMs was previously investigated, from a theoretical point of view, in [14], but only certain particular types of hierarchies were considered. This paper generalizes these previous results. The application of this technique to hierarchies with history states is also discussed in the paper. The method is then adapted for testing a system of communicating FSMs, in which the communication is one-directional, from one master to one or more slaves.

The paper is structured as follows. Section 2 introduces the FSM model and briefly presents the *W*-method. The technique for generating tests from hierarchical FSMs is presented in Section 3, while its application to hierarchies with history states is discussed in the next section. A case study to illustrate the development and testing of hierarchical FSMs is given in Section 5. The method adapted for a system of communicating FSMs is presented in Section 6. An analytical evaluation of the proposed approach is given in Section 7 and related work is discussed in the next session. Finally, conclusions are drawn in Section 9.

2. FINITE STATE MACHINES

First, we introduce basic FSM concepts and results and briefly describe the *W*-method.

An FSM M consists of a finite set of states Q, of which one is the designated initial state q_0, and transitions between states are labelled by pairs of input/output symbols from a finite input alphabet X and a finite output alphabet Y, respectively [4]. FSMs are usually represented as state-transition diagrams. For example, the FSM model of a simple tape recorder is given in Fig. 1.

The FSMs referred to in this paper are assumed to be *deterministic*. In such an FSM, for any given state q and any given input x, there is at most one transition from q labelled by x. When there is exactly one such transition, the FSM is said to be *completely specified*; otherwise, it is said to be *partially specified*. An FSM may be transformed into one that is completely specified by assuming that the 'refused' inputs produce a designated error output, which is not in the output

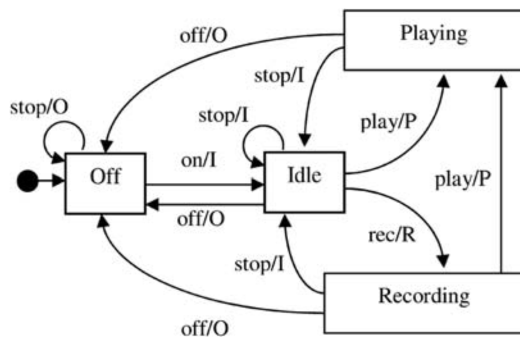

FIGURE 1. FSM model of a tape recorder.

alphabet of M; the erroneous behaviour can be represented as self-looping transitions or transitions to an extra (error) state [4]. However, in this paper we discuss the more general case when FSMs may be partially specified.

Given a state q, the finite sequences of input/output symbols that can be traced out of q make up the language accepted by M in q, denoted $L_M(q)$ or simply L_M when q is the initial state q_0. When only input sequences are of interest, the language will be denoted $Li_M(q)$ or Li_M, respectively.

Once we have an FSM representation of a system, we can use appropriate techniques to uncover possible errors in the implementation, such as erroneous transition labels, erroneous next-states, missing states, extra states, etc. One of the most general approaches is the *W*-method [10], which generates sequences of input symbols to (1) reach every state in the diagram, (2) check all the transitions from that state and (3) identify all destination states to ensure that their counterparts in the implementation are correct. Consider, for example, the FSM representation of a tape recorder given earlier and take the `Playing` state. This is reached from the initial state, `Off`, by a sequence of input symbols, $s_{\text{Playing}} =$ `on play`. All possible transitions emerging from this state need to be checked; in order to achieve this, each input symbol is applied after s_{Playing}. Furthermore, the original state, `Playing`, as well as the next-states of all these transitions, will have to be identified, so the resulting sequences will be, in turn, concatenated with each element of a set W that distinguishes between every pair of states. Therefore, the *W*-method involves the construction of two sets of input sequences:

- A *state cover* $S \subseteq X^*$ that reaches every state of the FSM; in particular, the empty sequence ϵ, which reaches the initial state q_0, is contained in S.
- A *characterization set* $W \subseteq X^*$ that distinguishes between every pair of states in the FSM. In other words, for every two distinct states q_1 and q_2, W contains at least one input sequence that produces different outputs when applied from q_1 and q_2, respectively. When the FSM is partially specified, this includes the case in which the input sequence can be applied in q_1 but not in q_2 or vice-versa.

An FSM in which all states are reachable and pairwise distinguishable is called *minimal*. As a state cover and a characterization set are assumed to exist, a prerequisite of the method is that the FSM specification is minimal. However, this condition does not restrict the applicability of the *W*-method since for every FSM a minimal, functionally (i.e. input/output) equivalent FSM can be constructed [4].

For the FSM represented in Fig. 1, ϵ reaches `Off`, `on` reaches `Idle`, `on play` reaches `Playing` while `on rec` reaches `Recording`. Thus $S = \{\epsilon,$ `on`, `on play`, `on rec`$\}$ is a state cover. On the other hand, the input symbol `off` can be applied in `Idle`, `Playing` and `Recording` but not

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

in Off, so off distinguishes the state Off from any of the remaining states. Similarly, play distinguishes Playing from Idle and Recording. Finally, rec distinguishes between Idle and Recording. Thus $W = \{$off, play, rec$\}$ is a characterization set.

Suppose we have an FSM specification M and we have constructed a state cover S and a characterization set W of M. Naturally, we assume that the implementation of M can be modelled by an unknown FSM, M'. The only information we need about M' is an estimation of the maximum possible number of states n' that it may have. This is based on the tester's knowledge of how the system has been implemented. Suppose we denote by k, $k \geq 0$, the difference between n' and the number of states n of the specification M. Then a test suite T can be constructed by first concatenating S with the set $X[k+1] = X^{k+1} \cup \ldots \cup \{\epsilon\}$ of all input sequences of length at most $k+1$, and then concatenating the resulting set with W. Thus

$$T = SX[k+1]W.$$

Note that the above formula is given in [10] for the case in which the FSM specification is completely specified. As explained in [15], when the specification is partially specified, the test suite will also include some of the prefixes of the sequences in T. However, in order to check the result produced by an input sequence, one would normally check the results produced by all its prefixes; consequently, this extra sub-set of prefixes does not explicitly appear in the formula.

The idea is that the set $S\,X[1] = S \cup S\,X$ (usually called a *transition cover* of M) ensures that all the states and all the transitions in M are also present in M', and $X[k]\ W$ ensures that M' is in the same state as M after performing each transition. Note that the latter set contains W and also all sets X^iW, $1 \leq i \leq k$. This ensures that M', does not contain extra states. If there were up to k extra states, then each of them would be reached by some input sequence of length up to k from the existing states.

In general, a transition cover is derived from a transition tree constructed in a breadth-first fashion, while a characterization set is constructed by gradually partitioning the state set based on the responses produced by sequences of length $k \geq 1$, using the so-called P_k tables [10].

Variants of the W-method also exist in the literature. The *partial W-method* (or *W_p-method*) [16] is an improvement of the W-method that may reduce the size of the test suite at the expense of a slightly more complex generation algorithm. The *round trip* [17] approach is based on a transition tree constructed in a depth-first fashion.

A *precise oracle*, which dumps the values of state variables and compares them to what is expected, can be used instead of W to check the state of the implementation under test. On the other hand, when the states of the FSM are actually abstract states, obtained by suitably partitioning the domain of the system data—as in the case of a statechart model of a class, for example—a precise oracle may be too expensive and so an *abstract oracle* [18], which entails checking the invariants of the states that are expected to be reached during the execution of test cases, can be used instead.

3. TESTING HIERARCHICAL FSMs

So far we have considered states as single entities. When modelling more complex systems, however, it is often useful to consider one or more states as containing an internal behaviour that can be itself be represented as an FSM. Such states are usually referred to as *compound states* and the FSMs that describe their internal behaviour as *internal FSMs*. The resulting model is called a *hierarchical FSM*, as opposed to a *flat FSM*, for when the states have no internal behaviour.

Suppose, for example, that we want to add FastForward, Rewind and Pause facilities to the tape recorder described earlier. This can be achieved by representing the PLAYING, RECORDING, and IDLE states themselves as FSMs, as shown in Fig. 2. The transitions entering a compound state will actually lead to the initial state of the corresponding internal FSM, whereas the transitions leaving a compound state will correspond to transitions leaving each state of the internal FSM. For example, the stop/I transition between compound states PLAYING and IDLE is equivalent to two transitions, from each of the internal states Play and PausePlay of PLAYING to the initial internal state Stop of IDLE. Similarly, the loop-back transition stop/I from IDLE is equivalent to three transitions, from internal states Stop, FF

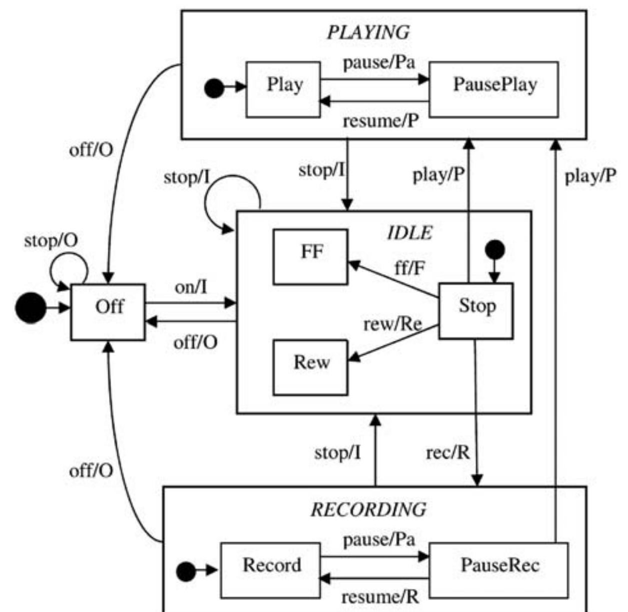

FIGURE 2. Hierarchical FSM for a tape recorder.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

and Rew to the initial internal state Stop. Furthermore, transitions from internal states to external states may also exist, e.g. the play/P transition from Stop to PLAYING. Using the afore-mentioned rules, it is possible to flatten a hierarchical FSM, that is to turn it into a behaviourally equivalent flat FSM.

There are two ways we can handle test generation from a hierarchical FSM. We can either generate tests solely on the basis of the final, flattened, specification or we can have the implementation developed in parallel with the specification, through a process of refinement, and reuse the test suites for the previous versions in the construction of the final test suite. Consider again the tape recorder example. Suppose the original specification represented in Fig. 1 is implemented first and then the implementations of the internal FSMs are 'placed' in the appropriate states. Then the implementation of the overall system can also be modelled by a hierarchical FSM.

More generally, suppose we have built a hierarchical FSM specification by representing one or more states of an original, flat, FSM M as internal FSMs. The set of all such (compound) states is denoted by Q_c, for each state $q \in Q_c$, the corresponding internal FSM is denoted by M_q. The input alphabets of M and M_q are denoted by X and X_q, respectively. The internal and external input alphabets are disjoint (otherwise the system may exhibit non-deterministic behaviour), so for every $q \in Q_c$, $X \cap X_q = \emptyset$.

As previously discussed, the hierarchical FSM preserves the original functionality described by M, but adds further detail. Consequently, every transition x/y from state q_1 to state q_2 in M will be replaced by one or many x/y transitions in the hierarchical FSM. More precisely, the following two cases can be distinguished:

- If $q_1 \notin Q_c$ (i.e. q_1 does not become a compound state in the hierarchical FSM) then the original transition will be replaced in the hierarchical FSM by *exactly one* transition, from q_1 to the initial state of M_{q_2}. Note that the case in which $q_2 \notin Q_c$ (i.e. q_2 does not become a compound state) is a particular case of this rule; in this situation, M_{q_2} is the 'trivial' FSM, consisting only of one state and no transitions, so the corresponding transition in the hierarchical FSM is from q_1 to q_2.
- If $q_1 \in Q_c$ then the original transition will be replaced in the hierarchical FSM by *at least one* transition, from one or more states of q_1 to the initial state of M_{q_2}. As explained above, the case $q_2 \notin Q$ is a particular case of this rule.

If the above condition is satisfied we say that the hierarchical FSM is a *refinement* of M.The equivalent, flattened, FSM of the hierarchical specification is denoted by M'. As all states of the internal FSMs are reachable (all internal FSMs are assumed to be minimal), each transition in M will be 'refined' into one or more sequences of transitions in M'.

Suppose that M and M_q have been implemented; the models of their implementations are denoted by M_I and M_{qI}, respectively. Suppose also that the whole implementation can be modelled by an (unknown) hierarchical FSM, in which the FSMs M_{qI} are placed within the states of M_I. As in the case of the specification, the hierarchical model of the implementation is a refinement (in the sense indicated above) of the original model. Analogously to the W-method, we assume that we know the difference between the estimated maximum number of states of the implementation model and the number of states of the specification for M and M_q, $q \in Q_c$; these are denoted by k and k_q, respectively. Finally, suppose that we have generated a test suite T for M and test suites T_q for M_q, $q \in Q_c$. Then, a test suite T' for M' can be constructed as the union of three sets, T_1, T_2 and T_3: T_1 will test the implementation of the original FSM M, T_2 will test the implementations of the internal FSMs M_q, while T_3 will check the integration of these implementations.

The first set T_1 is constructed by 'refining' the sequences in the original test suite T. The procedure is outlined below.

- Initialize $T_1 = T$.
- For every sequence $t \in T_1$ and for every compound state $q \in Q_c$ such that t crosses q (one or more times), replace each transition from q with one of the corresponding sequences of transitions from the hierarchical FSM. Note that the definition of refinement for FSMs given earlier ensures that at least one such sequence will exist. Naturally, when many corresponding sequences exist, the shortest will normally be selected.

Suppose, for example, that $t =$ on rec play is a test sequence for the original tape recorder model. Since on enters the initial internal state, Stop, of IDLE and rec leaves the same internal state, there is no need to insert any transfer sequence between these two input symbols. On the other hand, rec enters the initial internal state, Record, of RECORDING, while play will leave PauseRec. Consequently, a sequence that performs the transfer between Record and PauseRec, for example pause, will have to be inserted between rec and play. Thus the resulting, refined, test sequence is $t' =$ on rec pause play.

In order to test the implementation of an internal FSM M_q, we simply need to reach its initial state and apply the test suite T_q. The procedure is described in what follows:

- For each $q \in Q_c$, select an input sequence $p_q \in X^*$ that reaches q. Refine p_q by replacing (if necessary) transitions from the original FSM with the corresponding sequences of transitions from the hierarchical machine (as in the construction of T_1); the resulting sequence is denoted by p'_q. In our example, $p_{\mathrm{Idle}} = p'_{\mathrm{Idle}} =$ on, $p_{\mathrm{Playing}} = p'_{\mathrm{Playing}} =$ on play, $p_{\mathrm{Recording}} = p'_{\mathrm{Recording}} =$ on rec.
- Concatenate each p'_q with T_q. Thus $T_2 = \cup_{q \in Q_c} \{p'_q\}\ T_q$.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

The construction of the set T_3, which checks the integration of the internal implementations within the system, is slightly more complex. From the system passing all tests in the set T_1 we can deduce that M_I is functionally equivalent to M. However, M_I may not be minimal—unlike the specification, the minimality cannot be enforced on the implementation. Consequently, if the difference k between the maximum number of states of M_I and the number of states of M is greater than 0, a state q of the specification may correspond to many, equivalent, states in the implementation. If q is a compound state, M_{qI} will be used to detail the behaviour of each of the corresponding states of the implementation, so the integration of M_{qI} within any such state needs be checked. The procedure for constructing T_3 is given below.

- Select a state cover S of M.
- Extend all sequences in S with sequences of length at most k and, for each $q \in Q_c$, select the set R_q of all sequences in $S\,X[k]$ that reach q. Since M and M_I have been shown to be equivalent, R_q will reach all states in M_I that are equivalent to q.
- For each $q \in Q_c$, refine R_q (by replacing transitions from the original specification to sequences of transitions from the hierarchical FSM, as explained above); the resulting set of sequences is denoted by R'_q.
- For each $q \in Q_c$, select a state cover S_q of M_q (the same state cover used in the construction of T_q will normally be used). Since the system has passed all tests in T_2, M_{qI} is already known to be equivalent to M_q, so the sequences in S_q will reach a number of states of M_{qI} equal to the number of states of M_q. Then, if we extend each path in S_q with sequences of length at most k_q, the resulting set of valid paths, $P_q = S_q X_q[k_q] \cap Li_{M_q}$ will reach all states of M_{qI} (see [19] for a proof). Therefore, $P_q\ X$ will check all transitions from the (internal) states of M_{qI} to external states.
- Finally, all we have to do is to concatenate every sequence in R'_q with $P_q\ X$. Thus $T_3 = \cup_{q \in Q_c} R'_q\ P_q\ X$.

In our example, consider q = `Idle` and suppose $k = 1$ and $k_{\text{Idle}} = 1$. $X = \{$`on`, `off`, `stop`, `play`, `rec`$\}$ and $X_{\text{Idle}} = \{$`ff`, `rew`$\}$ are the input alphabets and $S = \{\epsilon$, `on`, `on play`, `on rec`$\}$ and $S_{\text{Idle}} = \{\epsilon$, `ff`, `rew`$\}$ are state covers. $R_{\text{Idle}} = \{$`on`, `on stop`, `on play stop`, `on rec stop`$\}$ is the set of all sequences in $S \cup S\ X$ that reach *Idle* and $R'_{\text{Idle}} = R_{\text{Idle}}$. Finally, $P_{\text{Idle}} = \{\epsilon$, `ff`, `rew`$\}$ is the set of all valid paths in $S_{\text{Idle}} \cup S_{\text{Idle}}\ X_{\text{Idle}}$.

The above explanations that accompany the construction of the three test suites outline a soundness proof of the method. This is not formalized since it would make the paper much harder to read. Formal proofs for two particular cases are given in [14]. The two cases considered are (1) when, for each state q, every transition from q in the original machine is replaced in the hierarchical FSM with transitions from all states of M_q and (2) when there exists an 'escape sequence' e, constructed exclusively from internal input symbols, such that, for each state q, every transition from q in the original machine is replaced with at least one transition, from state q' of M_q, where q' is the internal state reached by applying e in the initial state of M_q. It can be observed that the hierarchical FSM for the tape recorder considered above is in neither of these two categories.

For multiple-level hierarchies (i.e. whose hierarchies in which some internal FSMs are themselves described by lower-level FSMs), the procedure will be applied bottom-up to gradually test each FSM component and integrate it within the upper-level FSM. As in the case of flat FSMs, precise or abstract oracles may be used instead of characterization sets to check the precise or abstract state, respectively, of the corresponding FSM.

4. TESTING HIERARCHIES WITH HISTORY CONNECTORS

Hierarchical FSMs may contain a special kind of state, called *history states* or *history connectors* [3]. A history connector, usually represented as a circled 'H', is attached to a compound state and remembers its last current internal state. In this paper, we only discuss the so-called *shallow history connector* [3], which remembers the last current internal state at the same hierarchy level, as opposed to a *deep history connector*, which remembers the last current leaf state at the lowest hierarchy level. However, the underlying ideas presented here also apply to deep history connectors.

Consider, for example, the hierarchical FSM model of a computer system consisting of a simplified word processor (that allows text editing, table and diagram insertion) and a screen saver, as represented in Fig. 3. The screen saver is

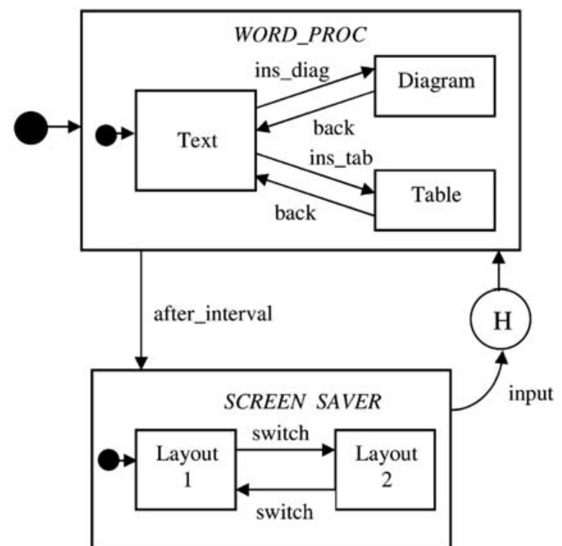

FIGURE 3. Hierarchical FSM with history connector.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

activated if the system has not received any input (i.e. key press or mouse click) for a certain time interval; the interrupted processing is then resumed when a new input arrives. The diagram is kept simple in order to illustrate more clearly the underlying ideas; the outputs produced by transitions, which do not play any part in our discussion, are ignored.

When the compound state `WORD_PROC` is entered for the first time, the processing will start from the initial internal state, `Text`. However, when the system subsequently re-enters the `WORD_PROC` state, after the screen saver is deactivated, the processing will have to be resumed from where it has left off and not always be restarted from the initial `Text` sub-state. This is specified by using the history connector attached to `WORD_PROC`. For the sake of discussion, we consider that the screen saver may have two different layouts and the user can switch between them; consequently, `SCREEN_SAVER` is also a compound state, but without a history connector.

A hierarchical state diagram with a history connector can be transformed into an equivalent hierarchical FSM without the connector by multiplying the external states of the diagram, so that distinct external states are created for each internal state which can be 'remembered' by the connector. For example, the equivalent model for the word processor/screen saver system will have three `SCREEN_SAVER` states, one for each sub-state of the word processor, as represented in Fig. 4. The equivalent model can then be used as a basis for test generation and the procedure presented in the previous section can be applied. It has to be noted that the multiplication of states associated with the construction of the equivalent model without connectors will increase the size of the 'integration' test suite T_3, but will not affect the 'internal' test suite T_2. In our example, it is sufficient to test the (internal)

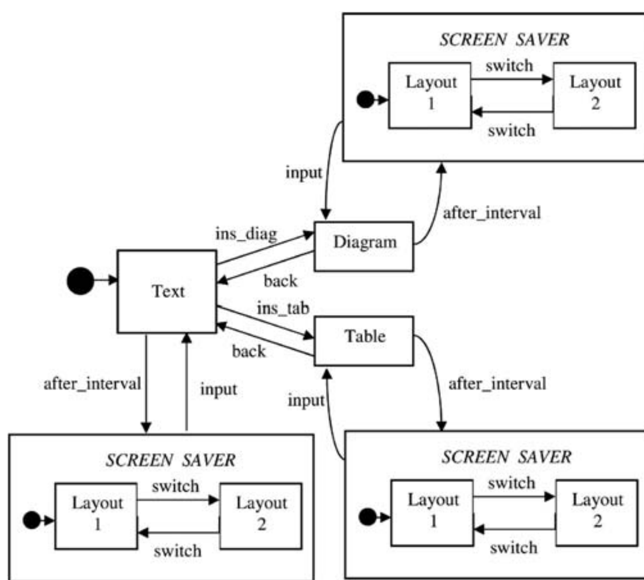

FIGURE 4. Equivalent model without history connectors.

`SCREEN_SAVER` FSM once, but, since it is used in three distinct places, its integration within the system will need to be tested three times, once for each state of the word processor.

On the other hand, since the equivalent model (without connectors) may have a significantly larger number of states than the original, it would be tempting to avoid its construction and generate test sets directly from the original model, by separating the testing of the underlying hierarchy from the testing of the history connector itself. Again, tests for the hierarchical FSM can be derived using the above procedure. As suggested in [20], the implementation of the history connector can be tested by entering and exiting the corresponding internal diagram, entering it again and verifying the entered state. While this approach may yield a significantly smaller test suite, it is usually unrealistic since it works on the assumption that there is an actual implementation of the history connector which can be identified and tested separately. This is not normally the case. The history state reflects an external observer's (designer's) view of the behaviour and not the implementer's view of behaviour. A history connector is normally implemented by passing information about the last-visited internal state to a global variable which is then read when the internal FSM is re-entered. Consequently, the states of the internal FSM are externalized through the use of a global variable. Thus, realistic tests for such an implementation can only be derived from an equivalent model of the type discussed earlier.

5. CASE STUDY: USING A HIERARCHICAL FSM TO SPECIFY A WORD PROCESSOR

Hierarchical FSMs provide a simple but effective means of gradually developing specifications for complex systems. We illustrate the approach on a word processor. For the sake of clarity, the functionality considered is fairly basic, but a much more complex word processor can be specified in this way. In order to keep the diagrams simple, the outputs produced by transitions are omitted.

At the first level in the hierarchy (Fig. 5) we look at the main functionalities of the system and decide which require separate states. Suppose that, apart from normal text editing, the system also allows the user to work with tables and equations and to draw pictures. In order to use these additional facilities, the user must select the appropriate option (e.g. `ins_draw`, `ins_tab`, etc.), perform the corresponding operation (which is not described at this level) and then either successfully finalize the operation (`ok`) or cancel it (`cancel`).

Each of these operations are then detailed at the next level of the hierarchy, by replacing each state with an appropriate internal FSMs. Some of these internal FSMs may only consist of one state and the required operations (e.g. normal editing will not involve additional, internal, states so it will be sufficient to add a loop-back transition in the `Normal`

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

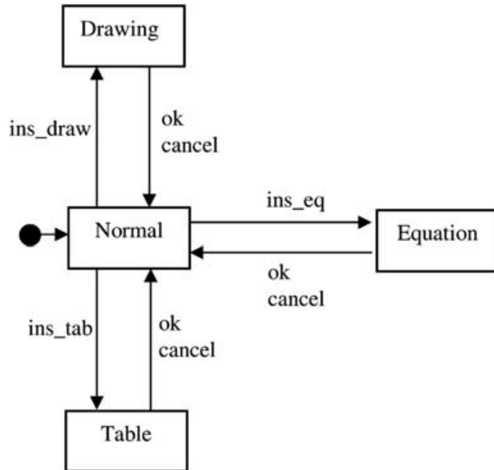

FIGURE 5. First-level specification for a word processor.

state), others may be more complex. Suppose, for example, that in order to draw a picture the user will be required to choose its shape (sel_shape), set its position (set_position) and, finally, adjust (i.e. move and resize) the selected shape. This functionality is described by the DRAWING internal FSM in Fig. 6. The 'adjusting' operation is not described at this stage, it will be detailed at the next level of the hierarchy (Fig. 7). According to Fig. 7, the picture can be moved by first enabling the 'move' operation (enable_move) and then dragging the picture (drag); alternatively, the move operation can be aborted (abort). Similarly, the 'resize' operation can be performed by selecting (select_point) and dragging (drag) a point of the shape in question. In this manner, it is possible to gradually refine the system functionality until the required level of detail is reached.

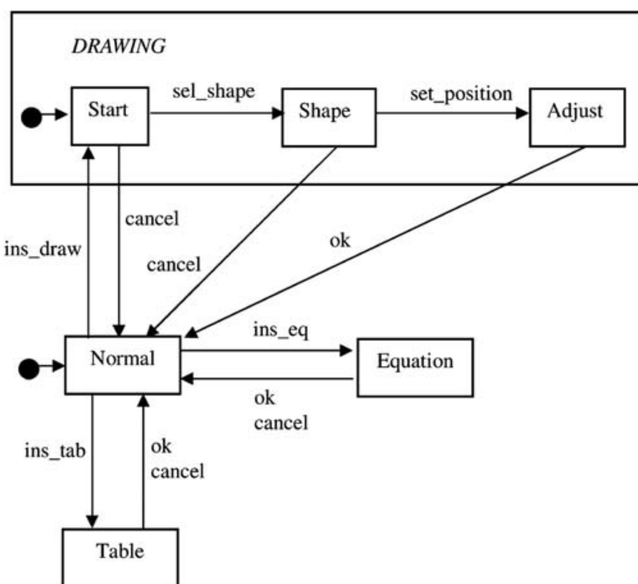

FIGURE 6. Second-level specification for DRAWING.

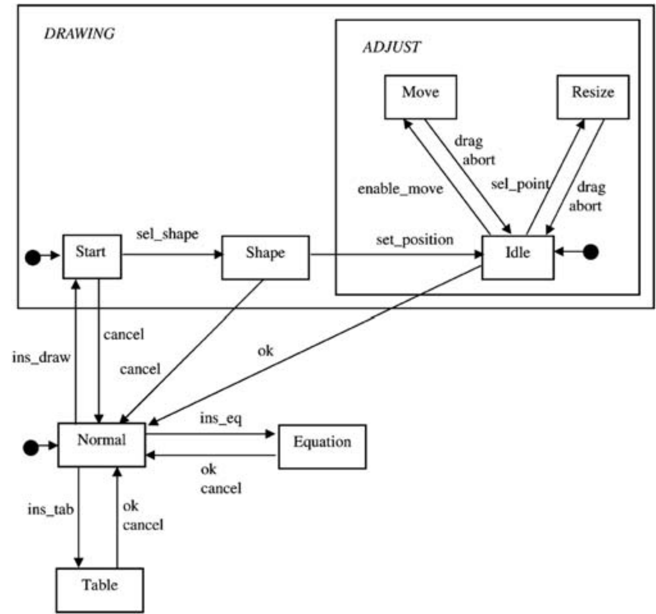

FIGURE 7. Third-level specification for ADJUST.

However, the case study also reveals a few limitations of the approach. Firstly, in the type of hierarchical FSMs considered in this paper, any transition leading to a compound state will always enter the initial state of the corresponding internal FSM. This may result in the duplication of some internal states. Suppose, for example, that the system allows not only the insertion of new drawings but also the adjustment of existing drawings. This functionality corresponds to the Adjust state of DRAWING in Fig. 6, which is then detailed by an internal FSM in Fig. 7. Clearly, rather than representing this functionality as a separate diagram, distinct from DRAWING, it would be tempting to reuse the DRAWING internal FSM and thus allow external transitions to enter a non-initial state of DRAWING, in this case Adjust. However, this approach would not suit our refinement-based testing philosophy. Consider an extended version of the word processor that also incorporates the 'modify drawing' functionality and suppose external transitions that lead to non-initial states of internal FSMs are permitted. At the first level in the hierarchy (Fig. 8), the ins_draw and modif_draw transitions from Normal will both lead to the same state, Drawing. Consequently, when test suites for the second level of the hierarchy (Fig. 9) are constructed, the DRAWING FSM and its integration within the external diagram will only be checked once (by the sets T_2 and T_3 in Section 3), either via the ins_draw transition or the modif_draw transition. This would not be sufficient since the behaviour exhibited by the DRAWING FSM in the state Adjust will differ from the behaviour exhibited by the same FSM in the initial state, Start. Furthermore, some of the external transitions of DRAWING may be accessible from its initial state, but not from other internal states (e.g.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

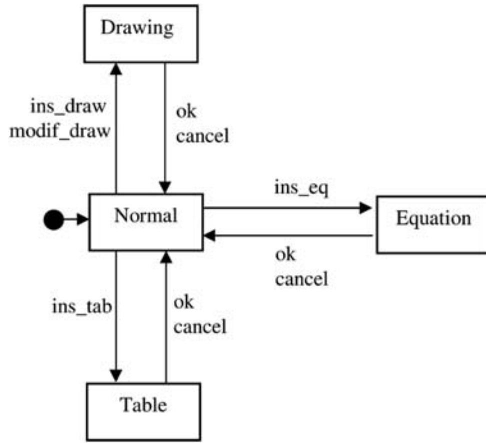

FIGURE 8. Incorrect first-level specification for the extended word processor.

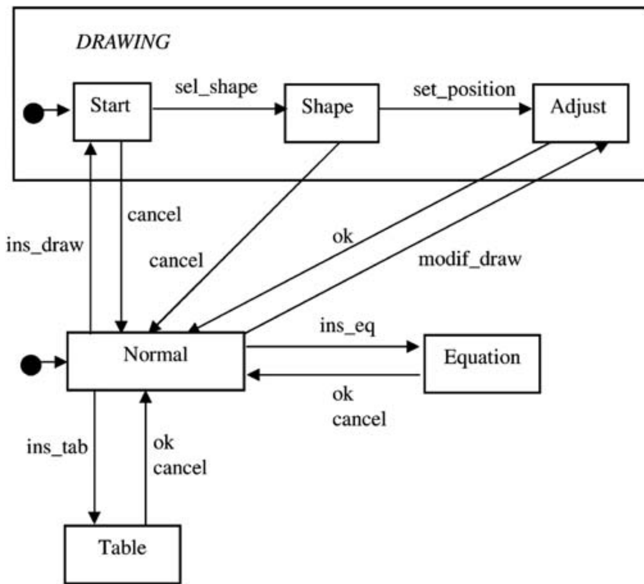

FIGURE 9. Incorrect second-level specification for the extended word processor.

no `cancel` transition is accessible from `Adjust`). Thus, the correct approach would be to represent `ins_draw` and `modif_draw` as transitions leading to different states (as in Fig. 10), which are then detailed as separate FSMs at the next level in the hierarchy (as in Fig. 11). Furthermore, test savings can still be achieved if the implementation of `DRAWING` is reused. In this case, this implementation will not have to be checked twice (in the construction of the test suite T_1 in Section 3); it will be sufficient to test it once, via the `ins_draw` transition (that leads to the its initial state) and then just check that the `modif_draw` transition leads to the correct state, `Adjust`—for this it is sufficient to concatenate `modif_draw` with a characterization set of `DRAWING`.

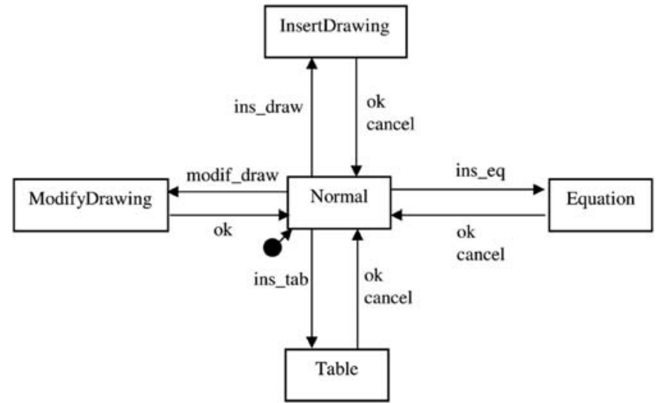

FIGURE 10. Correct first-level specification for the extended word processor.

Other limitations relate to the use of FSMs as a modelling tool, rather than being limitations of the hierarchical model itself. As the FSM model does not include a data structure, the level of abstraction is low and, consequently, different states may be required for essentially very similar things. For example, distinct `Shape` and `Adjust` states (and implicitly different `Idle`, `Move` and `Resize` states) will have to be created for different shape types that are moved or resized differently (e.g. a rectangle can only be dragged but a line can also be rotated). The level of abstraction of a specification can be increased by using *extended* FSMs such as stream X-machines [11, 19]. On the other hand, the transitions of an extended FSM will also depend on internal variables and so not all paths may be feasible; that is, there may be paths that cannot be driven by any sequence of inputs applied to the machine. However, stream X-machine-based, test-generation methods that can deal with this situation exist [15].

We now consider the application of our test-generation method to the hierarchical specification represented in Fig. 7. The input alphabets of the original specification (represented in Fig. 5), `DRAWING` internal FSM (see Fig. 6) and `ADJUST` internal FSM (see Fig. 7) are denoted by X, X_D and X_A, respectively. These alphabets are

- $X =$ {`ins_draw`, `ins_tab`, `ins_eq`, `ok`, `cancel`},
- $X_D =$ {`sel_shape`, `set_position`},
- $X_A =$ {`enable_move`, `sel_point`, `drag`, `abort`}.

The state covers and characterization sets used in the construction of the test suites are denoted by S, S_D, S_A and W, W_D, W_A, respectively. These are

- $S =$ {ϵ, `ins_draw`, `ins_tab`, `ins_eq`},
- $W =$ {`ok`} (Naturally, the outputs produced by the `ok` transitions from `Drawing`, `Table` and `Equation`, respectively, which are omitted from diagrams, must be mutually distinct. Thus, as no `ok` transition is defined from `Normal`, `ok` will distinguish between any pair of states in the original diagram.),

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

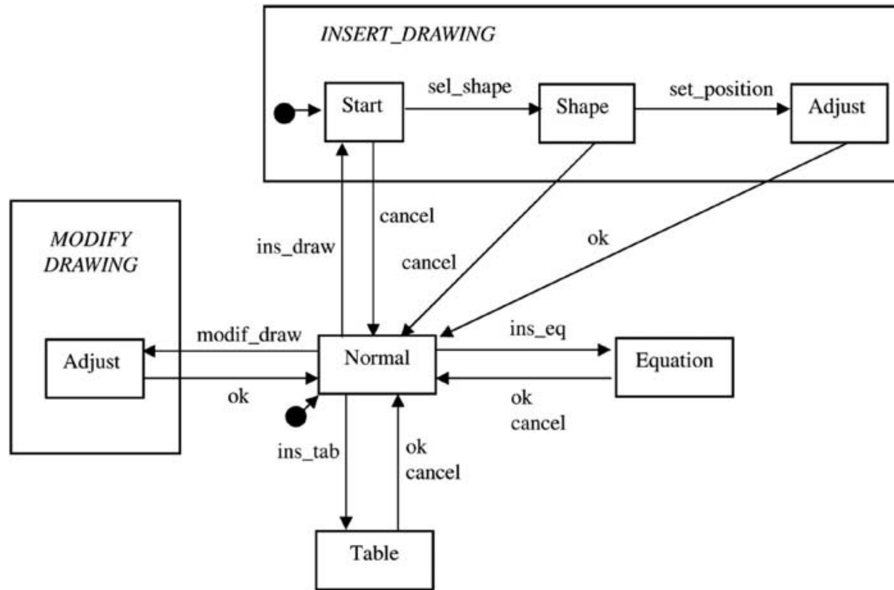

FIGURE 11. Correct second-level specification for the extended word processor.

- $S_D = \{\epsilon$, `sel_shape`, `sel_shape set_position`$\}$,
- $W_D = \{$`sel_shape`, `set_position`$\}$,
- $S_A = \{\epsilon$, `enable_move`, `sel_point`$\}$,
- $W_A = \{$`enable_move`, `sel_point`$\}$.

We work on the assumption that the difference between the estimated number of states of the implementation model and the number of states of the specification is 0 for all FSMs involved (the original FSM, the DRAWING and ADJUST internal FSMs). Thus, the test suites produced by the W-method for these three FSMs are

- $T = S\ X[1]\ W = W \cup X\ W \cup \{$`ins_draw`, `ins_tab`, `ins_eq`$\}\ X\ W$,
- $T_D = S_D\ \ X_D[1]\ \ W_D = W_D \cup X_D W_D \cup \{$`sel_shape`, `sel_shape set_position`$\}\ X_D\ W_D$,
- $T_A = S_A X_A[1]\ \ W_A = W_A \cup X_A W_A \cup \{$`enable_move`, `sel_point`$\}\ X_A\ W_A$.

Then the three test sets T_1^2, T_2^2, T_3^2 for the second-level specification (Fig. 6) are constructed as follows:

- T_1^2 is obtained from T by replacing each occurrence of the sequence `ins_draw ok` with the sequence `ins_draw sel_shape set_position ok`,
- $T_2^2 = \{$`ins_draw`$\}\ T_D$,
- $T_3^2 = \{$`ins_draw`$\}\ S_D\ X$.

The overall test suite for the second-level specification is $T^2 = T_1^2 \cup T_2^2 \cup T_3^2$. Finally, the three test sets for the third-level specification (Fig. 7) are obtained as follows:

- $T_1^3 = T^2 = T_1^2 \cup T_2^2 \cup T_3^2$,
- $T_2^3 = \{$`ins_draw sel_shape set_position`$\}\ T_A$,
- $T_3^3 = \{$`ins_draw sel_shape set_position`$\}\ S_A$ (X $\cup\ X_D$).

Thus, the resulting test suite for the entire hierarchy is $T_{\text{hier}} = T_1^3 \cup T_2^3 \cup T_3^3$.

By contrast, consider the case in which the test suite is derived directly from the equivalent, flattened, FSM of the final hierarchical specification. A state cover of the flattened specification is $S_{\text{flat}} = S \cup \{$`ins_draw`$\}\ S_D \cup \{$`ins_draw sel_shape set_position`$\}\ S_A$, while a characterization set is $W_{\text{flat}} = W \cup W_D \cup W_A$. The input alphabet of the flattened FSM is $X_{\text{flat}} = X \cup X_D \cup X_A$. Thus

- $X_{\text{flat}} = \{$`ins_draw`, `ins_tab`, `ins_eq`, `ok`, `cancel`, `sel_shape`, `set_position`, `enable_move`, `sel_point`, `drag`, `abort`$\}$,
- $S_{\text{flat}} = \{\epsilon$, `ins_draw`, `ins_tab`, `ins_eq`, `ins_draw sel_shape`, `ins_draw sel_shape set_position`, `ins_draw sel_shape set_position enable_move`, `ins_draw sel_shape set_position sel_point`$\}$,
- $W_{\text{flat}} = \{$`ok`, `sel_shape`, `set_position`, `enable_move`, `sel_point`$\}$.

The resulting test suite is $T_{\text{flat}} = S_{\text{flat}}\ X_{\text{flat}}[1]\ W_{\text{flat}} = W_{\text{flat}} \cup X_{\text{flat}}\ \ W_{\text{flat}} \cup \{$`ins_draw`, `ins_tab`, `ins_eq`, `ins_draw sel_shape ins_draw sel_shape set_position`, `ins_draw sel_shape set_position enable_move`, `ins_draw sel_shape set_position sel_point`$\}\ X_{\text{flat}}\ W_{\text{flat}}$.

Finally, let us compare the size (number of sequences) of the two test suites produced above. The number of elements of the input alphabets X, X_D and X_A are 5, 2 and 4, respectively.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

The transition covers S, S_D and S_A have 4, 3 and 3 elements, respectively, while the characterization sets W, W_D and W_A have 1, 2 and 2 elements, respectively. Thus, the test sets generated by the W-method, T, T_D and T_A have $1+5\cdot 1+3\cdot 5\cdot 1=21$, $2+2\cdot 2+2\cdot 2\cdot 2=14$ and $2+4\cdot 2+2\cdot 4\cdot 2=26$ elements, respectively. The test sets for the second-level specification T_1^2, T_2^2, T_3^2 have 21, 14 and $3\cdot 5=15$ elements, respectively, and so T^2 has $21+14+15=50$ elements. The test sets for the third-level specification T_1^3, T_2^3, T_3^3 have 50, 26 and $3\cdot(5+2)=21$ elements, respectively. Thus, the overall test suite T_{hier} produced by our method will contain $50+26+21=97$ sequences. On the other hand, X_{flat}, S_{flat} and W_{flat} have 11, 8 and 5 elements, respectively, and so T_{flat} will contain $5+11\cdot 5+7\cdot 11\cdot 5=445$ sequences.

6. TESTING MASTER–SLAVE COMMUNICATING FSMs

FSMs can also be used to model concurrent processes. When these processes communicate with each other, a slightly more complex model, called a *communicating finite state machine (CFSM)* [13] is needed. Basically, a CFSM is an FSM plus an input FIFO (first-in and first-out) queue; the CFSM only consumes the inputs from the queue. A *system of CFSMs* $(M_1, \ldots, M_n)$ works as follows:

- An input symbol x received from the external environment will go to the input queue of a CFSM, say M_i, provided it is contained in the input alphabet of M_i. In this paper, the input alphabets of the CFSMs are assumed to be disjoint, so x will enter one of the queues in a deterministic fashion.
- An output symbol y produced by a CFSM, say M_i, will pass to the input queue of another CFSM, say M_j, provided y is included in the input alphabet of M_j. If no such M_j exists, then x will go to the output environment.

A system of CFSMs is normally assumed to run in a *slow environment* and to have no live-lock [13]. We say that a system runs in a slow environment if inputs can be sent from the environment to the system only when the input queues of all CFSMs are empty. We say that a CFSM has a *live-lock* if it is possible to execute an infinite number of transitions without further inputs.

Consider, for example, the behaviour of an alarm radio modelled by a system of two CFSMs, as represented in Fig. 12. The CFSM at the top, representing the alarm side of the process, sends a `switch` signal to the other CFSM, representing the radio, each time it senses the `starttime` or `endtime` event of the alarm; as a result, the radio will switch from `Idle` to `Play` or vice-versa. The outputs produced by the radio will be released to the output environment; for simplicity, a two-valued output is used: 1, for when the radio is playing and 0, for when it is not.

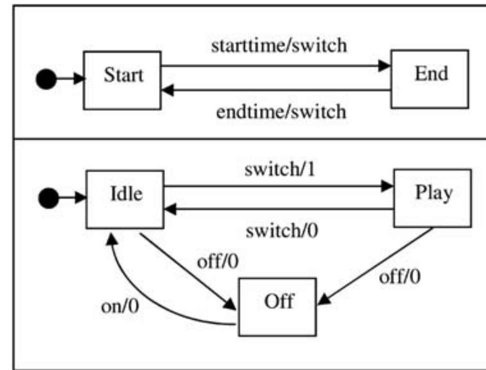

FIGURE 12. System of CFSMs for an alarm radio.

If a system of CFSMs runs in a slow environment and has no live-lock, then it can be turned into a behaviourally equivalent FSM whose (global) state set is the product of the state sets of the CFSM components [13]. As this equivalent product FSM suffers from the state explosion problem [4], the test suites derived from it may be unmanageably large. Consequently, whenever the particular situation allows it, alternative test-generation strategies are sought.

One case of practical interest is when the system is made of two CFSMs and the communication is one-directional: one of the CFSM components, say M_1, sends messages to the other, M_2, but not vice-versa. This master–slave type of interaction[1] between M_1 and M_2 can be described by a hierarchical model, in which the states of M_1 (the external FSM) become compound states and the behaviour of each is represented by M_2 (the internal FSM). The equivalent hierarchical model of the radio alarm system is given in Fig. 13. There are, however, differences between the hierarchical FSM model presented earlier and the hierarchical equivalent of a system of two CFSMs: the output produced by an external transition (of M_1) may be passed as input to the internal FSM, in which case the next internal state will depend on the input received (instead of always being the initial state of M_2, as in the hierarchical model presented earlier); furthermore, the next internal state may also depend on the internal state from which the external transition departs. Suppose, for example, that the `starttime/switch` transition leaves the `Idle` internal state of the compound state `START`. The `switch` output produced by the external transition will cause M_2 to move from `Idle` to `Play`, so the equivalent overall transition will enter the `Play` internal state of `END`. Similarly, when `starttime/switch` leaves the `Play` internal state of `START`, the destination internal state of `END` will be `Idle`.

[1]In a typical master–slave communication, the slave also sends a response, which causes the control to be transferred back to the master. In a CFSM system, however, this response is implicit: after the slave has performed the transition triggered by the symbol received from the master, the control is automatically transferred back to the master.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

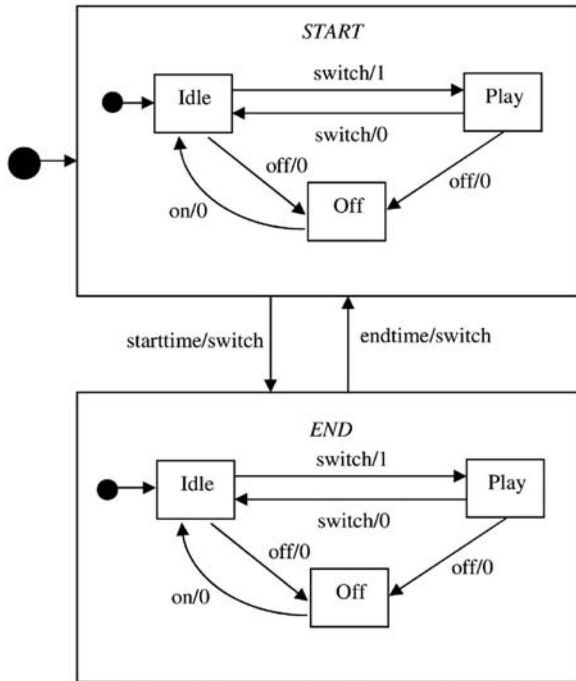

FIGURE 13. Hierarchical model for an alarm radio.

With this observation in mind, the test-generation procedure given earlier can be adapted for a system of two CFSMs, M_1 and M_2, in which the communication is one-directional, from M_1 to M_2. Analogously to the original procedure, three sets of test sequences, T_1, T_2 and T_3, will be produced:

(1) T_1 is the test suite for M_1. Since all external transitions (of M_1) can be triggered from any internal state (of M_2), and in particular from the initial internal state, no transfer sequences between internal states are needed.
(2) T_2 is the test suite for M_2. The test sequences for M_2 can be applied in the initial compound state (of M_1), so T_2 need not be prefixed with any transfer sequence.
(3) The construction of the integration test suite T_3 will take into account that the internal destination state of an external transition may depend on the previous internal state. Then the construction of T_3 involves the steps listed below, where k_1 and k_2 denote the difference between the maximum number of states of the implementation model and the number of states of the specification for M_1 and M_2, respectively:

- Select a state cover S_1 of M_1. In our example, $S_1 = \{\epsilon;$, `starttime`$\}$.
- Extend all sequences in S_1 with sequences of length at most k_1, and select the set $R = S_1X_1[k_1] \cap LiM_1$ of valid paths of M_1. As pointed out earlier, R need not be refined since the empty transfer sequence can be used. In our example, if $k_1 = 1$ then $R = \{\epsilon$, `starttime`, `starttime endtime`$\}$.
- For each sequence $r \in R$, let $q_2(r)$ be the state reached by M_2 when r is applied to the system. In our example, `starttime` will trigger the `switch`/1 transition in M_2, while `starttime endtime` will trigger a sequence of transitions, `switch`/1 `switch`/0. Thus $q_2(\epsilon) =$ `Idle`, $q_2($`starttime`$) =$ `Play`, $q_2($`starttime endtime`$) =$ `Idle`.
- For each sequence $r \in R$, select a state cover $S_{q2(r)}$ of M_2 from state $q_2(r)$. A state cover S_q of FSM M from state q is a set of input sequences S_q such that for every state q' of M that is reachable from q, there exists $s \in S_q$ that reaches q'. Note that, when q is not the initial state of M, not every state of M may be reachable from q. In our example, $S_\text{Idle} = S_\text{Play} = \{\epsilon$, `switch`, `off`$\}$.
- Extend each input sequence in $S_{q_2}(r)$ with sequences of length at most k_2 and select $P_{q_2}(r) = S_{q_2}(r)\, X_2[k_2] \cap Li_{M_2}(q_2(r))$ the set of all valid paths of M_2 from $q_2(r)$. In our example, if $k_2 = 1$ then $P_\text{Idle} = P_\text{Play} = \{\epsilon$, `switch`, `off`, `switch switch`, `switch off`, `off on`$\}$.
- Finally, concatenate every sequence r in R with the corresponding $P_{q_2}(r)$ and X_1. Thus $T_3 = \cup_{r \in R} \{r\} P_{q_2}(r)\, X_1$.

The above procedure can be extended to a system of CFSMs with one master M_1 and slaves $M_2, \ldots, M_n$, $n > 2$, which do not interact with each other. Since the slaves run independently, they can be tested separately from one another. Thus, the three test suites produced by the procedure will have the following form:

- T_1 is the test suite for the master CFSM M_1.
- $T_2 = \cup_{2 \leq i \leq n} T_2^i$, where T_2^i is the test suite for M_i.
- $T_3 = \cup_{2 \leq i \leq n} T_3^i$, where T_3^i is the integration test suite for M_i, constructed as indicated above.

Furthermore, the procedure can naturally be extended to a system of CFSMs that behaves like a multi-level hierarchy of master–slave interactions.

A system of CFSMs can also be used to describe the internal behaviour of compound states in a hierarchical FSM [3]. In this case, the above procedure can generate tests for the compound states, which are then used as inputs in the construction of the test suite for the hierarchical model.

7. DISCUSSION

Traditional methods for test generation from FSMs [4, 7] or (UML) state diagrams [17] can only be applied to flattened specifications. In comparison, the divide and conquer strategy used in this paper can yield considerably reduced test suites.

Consider the case of an FSM hierarchy with one compound state. According to [10], for an FSM specification with n states

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

and r input symbols and for n', the estimated maximum number of states of the implementation under test, equal to n, the W-method will produce at most $n^2 \cdot r$ sequences. Then, for a hierarchy formed from an external FSM M with n states and r input symbols and an internal FSM M_q with n_q states and r_q input symbols and for n' and n'_q, the estimated maximum number of states of the corresponding implementations, equal to n and n_q, respectively, the upper bounds on the number of elements of the test sets T_1 and T_2 will be $n^2 \cdot r$ and $n_q^2 \cdot r_q$, respectively. In order to construct the "integration" set, T_3, each of the n_q sequences that form a state cover of M_q is concatenated with every input symbol of M, so $T3$ will contain $n_q \cdot r$ elements. Thus, the upper bound for the total number of sequences in $T_1 \cup T_2 \cup T_3$ will be $n^2 \cdot r + n_q^2 \cdot r_q + n_q \cdot r$. On the other hand, the maximum number of sequences produced by applying, under the same conditions, the W-method to the equivalent flat FSM will be $(n + n_q - 1)^2 \cdot (r + r_q)$. It can be observed that our method yields a considerably smaller test suite; for n, n_q, r and r_q reasonably large, the size difference can be substantial.

Consider also the case of a system composed of two CFSMs, a master M_1 and a slave M_2, having n_1 and n_2 states and r_1 and r_2 input symbols, respectively. This can be transformed into a behaviourally equivalent flat FSM (using the approach in [13]) with $n_1 \cdot n_2$ states and $r_1 + r_2$ input symbols. Thus, the maximum number of test sequences obtained by applying the W method to this flat specification will be $(n_1 \cdot n_2)^2 \cdot (r_1 + r_2)$. On the other hand, by using our approach, the system of two CFSMs will be represented as a hierarchy formed of an external FSM M_1 in which all states are replaced by the same internal FSM, M_2. The set T_1, which checks the external FSM, will have at most $n_1^2 \cdot r_1$ sequences, while the set T_2, which checks the internal FSM, will have at most $n_2^2 \cdot r_2$ sequences. The integration of M_2 within M_1 will be checked n_1 times, each time with $n_2 \cdot r_1$ sequences, so T_3 will contain $n_1 \cdot n_2 \cdot r_1$ sequences. Thus, an upper bound for the number of test sequences produced by our method will be $n_1^2 \cdot r_1 + n_2^2 \cdot r_2 + n_1 \cdot n_2 \cdot r_1$, which is much lower than the upper bound for the case in which test sequences are derived directly from the equivalent flat FSM.

8. RELATED WORK

Belli [21] applies the divide and conquer principle for reducing the complexity of tests derived from an FSM model of a GUI, but hierarchical FSMs are not explicitly considered. Andrews *et al.* [22] use constraints to reduce the set of input values and to help solve the state explosion problem in hierarchical FSM models of Web applications. Paiva *et al.* [23] exploit the structure of a hierarchical FSM to reduce the number of states in the equivalent flat FSM. However, neither [22] nor [23] approach the test-generation problem from the perspective of conformance testing, as in this paper; instead, test sequences are generated to achieve certain coverage criteria (e.g. node coverage, transition coverage, etc.). Bogdanov and Holcombe [20, 24] investigate test selection from hierarchical statecharts, in particular from Harel's semantics statecharts. Basically, two extreme cases are considered: when no restrictions are placed on the development of the implementation and when the specification is a 'geometrical hierarchy', so that faults related to not properly entering or exiting internal statecharts are ruled out by the construction of the system. In the former case, they provide, under certain constraints, formulas for constructing state covers and characterization sets for the equivalent (flat) statechart of the hierarchy from state covers and characterization sets of the external and internal statecharts. A test suite for the hierarchy can then be derived by simply applying the W-method. In the latter case, test generation for the overall system is reduced to test generation for the internal and external FSMs—basically, the sets T_1 and T_2 produced by our method. The former approach may produce large test suites and does not take advantage of the refinement strategy used in the design, while the latter has limited applicability; it cannot be used, for example, to generate tests for hierarchies in which external transitions leave some, but not all the sub-states of an internal FSM (e.g. in the tape recorder example, the `play/P` transition leaves `PauseRec`, but not `Record`). The method presented in this paper not only tests the implementation of the external and internal FSMs, but also the integration of these implementations, and so it can cope with a much richer hierarchy semantics. Bogdanov and Holcombe [20, 24] also investigate test selection from concurrent statecharts; the approach is similar to that used for test selection from hierarchical statecharts. Furthermore, the presented methods require communication to be disabled during testing, so, actually, they can only be applied to non-communicating concurrent statecharts. A similar approach is also used by Li and Qi [25] for testing hierarchical and communicating UML statecharts, but the test-selection strategies proposed are based on the Wp-method instead.

In general, a system of communicating FSMs is first transformed into a behaviourally equivalent (non-deterministic) FSM which is then tested using one of the established techniques, such as the W or Wp-methods [13]. While this approach is the most general and provides the best coverage, as pointed out earlier in the paper, it may suffer from a state explosion problem. Hierons [26] points out that for some communicating FSMs called *semi-independent communicating FSMs*, it is possible to test the core transition structure with transitions that do not communicate and then check the remaining ones separately. The execution of a communicating transition can lead to a sequence of transitions being executed, so the final states of these transitions need be checked. The problem of finding a set of sequences that minimize the total cost is shown to be NP-complete and possible heuristics are discussed in [26]. The test strategy employed is based on the

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

UIO method. The method presented in this paper, on the other hand, allows more general strategies, such as the *W* and *Wp*-methods, to be used for testing the individual FSMs.

9. CONCLUSIONS

In this paper, we present a method for generating tests for a hierarchical FSM by refining and reusing test suites for the components of the hierarchy. Test generation for FSM hierarchies with history states is also discussed. The method is also adapted for a system of communicating FSMs in which the communication is one-directional, from one master to one or more slaves. The proposed method is general, in the sense that the test suites for the individual components can be generated using any FSM-based technique.

A design is usually constructed gradually: initially, we may have only a skeleton, then step by step we fill it with details. Our approach is to develop the test sets in parallel with the design, rather than generating them only on the basis of the final specification. In this way, testing is kept consistent with the design and we avoid leaving testing to the final, pre-release, stage. Furthermore, by refining test suites in parallel with the specification and using a divide and conquer strategy, the size of the test data is considerably reduced in comparison with the case in which tests are derived directly from the final specification.

For testing to be efficient, it must be automated as much as possible. It is straightforward to convert an FSM into a computer program and this process can be easily automated. The test-generation procedures given in this paper, including the test selection for the individual FSMs (using the *W* or *Wp*-methods, for example), can be fully automated. Appropriate tools are under development.

Further work involves looking at test generation from systems of CFSMs with more complex types of communication and extending the method given here to extended FSM models, such as stream X-machines [11, 15]. Test generation from particular types of hierarchical stream X-machines has already been investigated in [19].

ACKNOWLEDGEMENTS

The author would like to thank the anonymous reviewers, whose comments have improved the presentation of this paper.

REFERENCES

[1] Harel, D. and Politi, M. (1998) *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*. McGraw-Hill, New York.

[2] ITU-T. (1999) *Recommendation Z.100 Specification and description language (SDL)*. International Telecommunications Union, Geneva, Switzerland.

[3] Booch, G., Rumbaugh, J. and Jacobson, I. (1999) *Unified Modelling Language User Guide*. Addison Wesley Longman.

[4] Lee, D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, **84**, 1090–1123.

[5] Ramalingam, T., Das, A. and Thulasiram, K. (1995) Fault detection and diagnosis capabilities of test sequence selection methods based on the FSM model. *Comput. Commun.*, **18**, 113–122.

[6] Ramalingam, T., Das, A. and Thulasiram, K. (1995) On testing and diagnosis of communication protocols based on the FSM model. *Comput. Commun.*, **18**, 329–337.

[7] Sidhu, D. and Leung, T. (1989) Formal methods for protocol testing: a detailed study. *IEEE Trans. Softw. Eng.*, **15**, 413–426.

[8] Shen, Y.-N., Lombardi, F. and Dahbura, A.T. (1990) Protocol Conformance Testing Using Multiple UIO Sequences. In Brinksma, E., Scollo, G. and Vissers, C.A. (eds), *Protocol Specification, Testing and Verification IX*, pp. 131–143. Elsevier Science B.V., North-Holland.

[9] Ural, H. (1992) Formal methods for test sequence generation. *Comput. Commun.*, **15**, 311–325.

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

[11] Holcombe, M. and Ipate, F. (1998) *Correct Systems: Building a Business Process Solution*. Springer, Berlin.

[12] Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H. and Cha, S.D. (1999) Test cases generation from UML state diagrams. *IEE Proc.*, **146**, 187–192.

[13] Luo, G., von Bochmann, G. and Petrenko, A. (1994) Test selection based on communicating non-deterministic finite-state machines using a generalised Wp-method. *IEEE Trans. Softw. Eng.* **20**, 149–162.

[14] Ipate, F. and Balanescu, T. (2005) Refinement in finite state machine testing. *Fundam. Inform.*, **64**, 191–203.

[15] Ipate, F. (2006) Testing against a non-controllable stream X-machine using state counting. *Theoret. Comput. Sci.*, **353**, 291–316.

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

[17] Binder, R.V. (1999) *Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology*. Addison-Wesley.

[18] Meyer, B. (1992) Design by contract. *IEEE Comput.*, **25**, 40–51.

[19] Ipate, F. and Holcombe, M. (2002) An integrated refinement and testing method for stream X-machines. *Appl. Algebra Eng., Commun. Comput.*, **13**, 67–91.

[20] Bogdanov, K. (2000) Automated testing of Harels statecharts. PhD Thesis, University of Sheffield.

[21] Belli, F. (2001) Finite State Testing and Analysis of Graphical User Interfaces. *12th Int. Symp. on Software Reliability Engineering (ISSRE 2001)*, Hong Kong, China, November 27–30, pp. 34–43.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011

[22] Andrews, A.A., Offutt, J. and Alexander, R.T. (2005) Testing Web applications by modeling with FSMs. *Softw. Sys. Model.*, **4**, 326–345.

[23] Paiva, A., Tillmann, N., Faria, J.C.P. and Vidal, R.F.A.M. (2005) Modeling and Testing Hierarchical GUIs. *Proc. 12th Int. Workshop on Abstract State Machines (ASM 2005)*, Paris, France, March 8–11, pp. 329–344.

[24] Bogdanov, K. and Holcombe, M. (2004) Refinement in statechart testing. *Softw. Test. Verif. Reliab.*, **14**, 189–211.

[25] Li, L. and Qi, Z. (1999) Test Selection from UML Statecharts. *TOOLS*, **31**, 273–281.

[26] Hierons, R.M. (1997) Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proc., Softw.*, **144**, 291–295.

Downloaded from comjnl.oxfordjournals.org at University of Pitesti on June 5, 2011