Using a SAT Solver to Generate Checking Sequences

Guy-Vincent Jourdan
SITE, Faculty of Engineering
University of Ottawa
Ottawa, Canada
gvj@site.uottawa.ca

Hasan Ural
SITE, Faculty of Engineering
University of Ottawa
Ottawa, Canada
ural@site.uottawa.ca

Hüsnü Yenigün
Sabanci University
Istanbul, TURKEY
yenigun@sabanciuniv.edu

Dong Zhu
SITE, Faculty of Engineering
University of Ottawa
Ottawa, Canada
dzhu063@uottawa.ca

Abstract—**Methods for software testing based on Finite State Machines (FSMs) have been researched since the early 60's. Many of these methods are about generating a checking sequence from a given FSM which is an input sequence that determines whether an implementation of the FSM is faulty or correct. In this paper, we consider one of these methods, which constructs a checking sequence by reducing the problem of generating a checking sequence to finding a Chinese rural postman tour on a graph induced by the FSM; we re-formulate the constraints used in this method as a set of Boolean formulas; and use a SAT solver to generate a checking sequence of minimal length.**

Keywords: Finite State Machines, Software Testing, Disinguishing Sequences, Checking Sequences, SAT Solvers

I. INTRODUCTION

Finite State Machine (FSM)-based specifications are often used as the basis for formal testing methods. This approach has been applied to a wide variety of systems such as telecommunications systems, communications protocols, pattern matching and machine learning. It consists of setting up a *fault detection experiment* [1], that is applying an input sequence derived from a specification FSM M to an implementation N of M and observing the output sequence produced by N in response to the input sequence. The actual output sequence is then compared to the expected output sequence. The applied input sequence is called a *checking sequence*. The checking sequence determines whether N is a correct or faulty implementation of M [2, 3].

Different methods for constructing a checking sequence from an FSM M use different concepts such as distinguishing sequence [2] and a set of characterizing sequences [4]. These methods target a fault model which is typically characterized by a set $\Omega(M)$ of potential implementations of FSM M with the same input and output alphabets as M and with at most the same number of states as M; and then for a given $N \in \Omega(M)$, recognize each distinct state in N as a distinct state of M and verify that each transition of M is correctly implemented in N. Some of these methods use a *distinguishing sequence* D, an input sequence for which the response of each state of M is distinct. In this case, the methods achieve recognition of a state of N as a state of M by applying D followed by a (possibly empty) input sequence which transfers M from one state to another state (this is called a *transfer sequence* T) at that state of N. In these methods, verification of a transition of M from state a to state b under input x in N is achieved in three steps 1) transferring N to the state recognized as state a of M; 2) checking the output produced by N in response to x to be as specified in M (to detect an *output fault*); and 3) recognizing the state reached by N after the application of x as state b of M (to detect a *transfer fault*). Step 1) is realized indirectly by making sure that the state reached by the application of a *DT at each state* is recognized by applying another D at some point in the checking sequence. This facilitates the use of a *DT* at a state such that the state reached by the application of *DT* is the state to which N needs to be transferred (i.e., state recognized as the starting state of a transition to be verified). Step 2) is realized by comparing the expected out with the actual output. Step 3) is realized by applying a *DT* at the state reached by N after the application of x.

A checking sequence of an FSM may be constructed as a concatenation of a set of paths, derived from the digraph representation of the FSM, which has three types of components: i) state recognition paths used to recognize each state of the FSM (e.g., α-sequences [6]); ii) transition verification paths used to verify each transition of the FSM (e.g., test segments); iii) transfer paths used to concatenate paths in i) and ii). Checking sequence generation methods place various constraints on the selection of transfer paths. Earlier methods use some predefined strategies to find transfer

This work has been supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, the Ontario Centers of Excellence, and Sabanci University

978-1-4244-5023-7/09/$25.00 ©2009 IEEE

paths as short as possible [3, 5] while constraints are enforced in the applied procedure. These strategies do not guarantee that transfer paths found yield minimized checking sequences. An optimization model has been proposed to solve this problem in [6] and it is adopted by some successive checking sequence generation methods [7, 8, 9, 10].

In this paper we suggest using a SAT solver to generate a checking sequence. A set of Boolean formulae encoding the requirements of a checking sequence is generated. Any satisfying assignment for the variables of the generated Boolean formulae indicates and corresponds to a checking sequence. A SAT solver works on this set of Boolean formulae to find such a satisfying assignment (i.e. a solution to the given SAT problem). Different checking sequence generation methods impose different conditions on the sequence to be generated. Therefore the set of Boolean formulae generated also depends on the method of generating a checking sequence. In this paper we use the method presented in [6].

The rest of the paper is organized as follows: Section II gives an overview of the method introduced in [6]. Section III provides all the encoding techniques that we have established in order to transform the method into a SAT problem. Section IV gives some technical details and application results. We survey related works in Section V, and we conclude the paper in Section VI.

II. Details of the Checking Sequence Constuction Method

A. Preliminaries

We represent a deterministic *finite state machine* (FSM) M as $(S, s_1, X, Y, \delta, \lambda)$, where S is a finite set of states with $n = |S|$, $s_1 \in S$ is the *initial state*, X is a finite set of inputs, Y is a finite set of outputs, δ is a state transition function that maps $S \times X$ to S and λ is an output function that maps $S \times X$ to Y. These two functions are extended to input sequences $I \in X^*$ in the usual manner. Below we give definitions from [11].

An FSM M is considered *minimal* if, for every pair of states $s_i, s_j \in S$, $i \neq j$, there is an input sequence $I \in X^*$ such that $\lambda(s_i, I) \neq \lambda(s_j, I)$. M is considered *completely specified*, if for each input $x \in X$ and for each state $s_i \in S$, $\delta(s_i, x)$ is defined. M can be represented by a *digraph* $G = (V, E)$ where a set of vertices V represents the set S of states of M, and a set of directed edges E represents all specified transitions of M. Each edge $e = (v_i, v_j; x / y) \in E$, is a state transition, denoted $t = (s_i, s_j; x/y)$, from state s_i to state s_j with input $x \in X$ and output $y \in Y$, where v_i and v_j are the *starting* and *terminating* vertices of e (states of t), and input/output (i.e., *i/o* pair) x/y is the label of e, denoted by *label*(e).

A *path* $P = (n_1,n_2; x_1/y_1)(n_2,n_3; x_2/y_2)\ldots(n_{r-1},n_r; x_{r-1}/y_{r-1})$, $r > 1$, of $G = (V, E)$ is a finite sequence of adjacent (not necessarily distinct) edges in E, where each node n_i, $1 \leq i \leq r$, represents a vertex of V; n_1 and n_r are called *starting* and *terminating* nodes of P, and the input/output sequence $(x_1/y_1)(x_2/y_2)\ldots(x_{r-1}/y_{r-1})$ is called *label* of P. P is represented by $(n_1,n_r; I/O)$, where I/O is the label of P, $I = x_1x_2\ldots x_{r-1}$ is called the *input portion* of I/O, $O = y_1y_2\ldots y_{r-1}$ is called the *output portion* of I/O. The input portion I of the label I/O of path $(n_1,n_r; I/O)$ will be called the *transfer sequence* T (from n_1 to n_r). The *length* of an *input sequence* I (or *input/output sequence* I/O) is its number of inputs, denoted by $|I|$ (or $|I/O|$).

Let $M = (S, X, Y, \delta, \lambda)$ denote a completely specified, minimal and deterministic FSM, which is represented by a strongly connected digraph $G = (V, E)$. Let the fault model for M, $\Omega(M)$, be the set of FSMs each of which has at most $n = |S|$ states and the same input and output sets as M. Let N be an FSM of $\Omega(M)$. N is *isomorphic* to M if there is a one-to-one and onto function f on the state sets of M and N such that for any state transition $(s_i, s_j; x / y)$ of M, $(f(s_i), f(s_j); x / y)$ is a transition of N. A *checking sequence* of M is an input sequence starting at the initial state s_1 of M that distinguishes M from any N of $\Omega(M)$ that is not isomorphic to M. (i.e., the output sequence produced by any such N of $\Omega(M)$ is different from the output sequence produced by M). That is to say, in response to this input sequence, any faulty implementation N from $\Omega(M)$ will produce an output sequence different from the expected output sequence, thereby indicate the presence of one or more faults.

For the method considered in this paper, the recognition of each distinct state in N as a distinct state of M and verification of whether each transition of M is correctly implemented in N are based on distinguishing sequences. A *distinguishing sequence* D of M is an input sequence such that the output sequence produced by M in response to D is different for each state of M (i.e., $\forall s_i, s_j \in S$, $s_i \neq s_j$; $\lambda(s_i,D) \neq \lambda(s_j,D)$). A distinguishing sequence D of an FSM M is then used as follows:

Consider a path P of G representing M. Let $Q = label(P)$.

1. A node n_i of P is *recognized* in Q as state s of M if
   a) n_i is the starting node of a subpath of P whose label is $DT / \lambda(s, DT)$ for some T or
   b) $(n_q, n_i; T / \lambda(s', T))$ and $(n_j, n_k; T / \lambda(s', T))$ are subpaths of P, n_q and n_j are recognized in Q as state s' of M, and node n_k is recognized in Q as state s of M.
2. A transition $t = (s_p, s_q; x / y)$ is *verified* in Q if there is a subpath $(n_i, n_{i+1}; x_i / y_i)$ of P such that n_i is recognized as s_p in Q, n_{i+1} is recognized as s_q in Q, $x_i/y_i = x / y$.

For the methods considered in this paper, if a path of G representing M starts from s_1, and verifies all transitions of M, the input portion of this path's label is a checking sequence of M [6].

B. The method of Ural, Wang and Zhang [6]

The method proposed by Ural, Wang and Zhang [6] first forms α-sequences as concatenations of $D/ \lambda(s_i, D)$ followed by a transfer sequence T_i at each state s_i until the application of the last $D/ \lambda(s_i, D)T_i$ is a repetition of an earlier application of $D/ \lambda(s_i, D)T_i$ in the path. The α-sequences are defined in the following way. The first step is to choose subsets V_k of V $(1 \leq k$

$\leq q$) whose union is V. The elements within each V_k are ordered giving $V_k=\{v_1^k, \ldots, v_{nk}^k\}$, where the state represented by v_i^k is denoted $s_{m(i,k)}$. For each v_i^k, we obtain a sequence $D/\ \lambda(s_{m(i,k)}, D)T_i^k$, which is the result of applying D in state $s_{m(i,k)}$ followed by a transfer sequence T_i^k whose final state corresponds to v_{i+1}^k (v_{nk+1}^k can be any v_w^k, $1 \leq w \leq n_k$). For each V_k we form a path P_k with starting state $s_{m(1,k)}$ and label $\alpha_k=D/\ \lambda(s_{m(1,k)}, D)T_1^k\ D/\ \lambda(s_{m(2,k)}, D)T_2^k \ldots\ D/\ \lambda(s_{m(nk,k)}, D)T_{nk}^k\ D/\ \lambda(s_{m(w,k)}, D)T_w^k$, $1 \leq w \leq n_k$. The set $A = \{\alpha_1, \ldots, \alpha_q\}$ is called an α-set and each sequence α_i in A is called an α-*sequence* from A. The transfer sequence, that follows the execution of D from state s_i, is denoted T_i.

Transition verification paths are formed by applying D after the transition's input. The method in [6] finds a shortest sequence containing all α-*sequences* and transition verification paths, possibly connected by transfer paths. A preset acyclic subset of transitions is chosen and the transfer paths are only allowed to contain transitions from this subset. It is proved that any sequence constructed in this way can be used to produce a checking sequence.

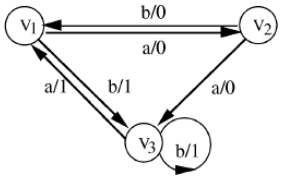

Figure 1: A sample FSM

We illustrate the method on the sample FSM shown on Figure 1. On this FSM, a distinguishing sequence is $D=ab$. There are two α-*sequences*, $\alpha_1=(DD)/(0000)$ and $\alpha_2=(DDD)/(011111)$. Note that in this example the transfer sequence T_i (following the application of D at state s_i) is taken to be the empty sequence for all states. The method first duplicates the set of vertices, creating a set $V'=\{v'_1, v'_2, v'_3\}$ duplicating the initial set $V=\{v_1, v_2, v_3\}$. It then creates several sets of edges between the elements of $V \cup V'$, in addition to the initial set E:

The first set of edges, E_α, is between V and V'. It connects the nodes of V to the node of V' via the α-*sequences*. In the case at hand, we have:

$$E_\alpha = \{(v_1, v'_1;\alpha_1), (v_2, v'_3;\alpha_2)\}.$$

The second set of edges, E_c, is between the elements of V', and is built with every outgoing transition (from the corresponding state in V), followed by D. In our example, we have:

$$E_c = \{(v'_1,v'_3;L_{123}),(v'_1,v'_3;L_{133}),(v'_2,v'_3;L_{233}),\ (v'_2,v'_1;L_{211}),(v'_3,v'_1;L_{311}),(v'_3,v'_3;L_{333})\},$$

where

$$L_{123}=(aD)/(001),\ L_{133}=(bD)/(111),\ L_{233}=(aD)/(011),\ L_{211}=(bD)/(000),\ L_{311}=(aD)/(100),L_{333}=(bD)/(111).$$

The third set of edges, E_T, is from V to V'. It is created by joining elements of V to elements of V' by applying D:

$$E_T = \{(v_1,v'_1;T_1),(v_2,v'_3;T_2),(v_3,v'_3;T_3)\},$$

where

$$T_1=(D)/(00),\ T_2=(D)/(01),\ T_3=(D)/(11).$$

A fourth set E'' (which is a copy of the subset of the edges from E), from V' to V' is created to ensure that the subgraph consisting of the vertices in V' and related edges is strongly connected. E'' can be as large as possible in order to give some flexibility to the algorithms that will be searching for a tour on this graph, but on the other hand the edges in E'' must not include any cyclic set of edges. In our case, we choose

$$E''=\{(v'_3, v'_1; a/1), (v'_1, v'_2; a/0)\}.$$

Finally, a fifth set E_ε is created, joining every vertex of V' to the corresponding vertex of V by an edge labeled ε:

$$E_\varepsilon = \{(v'_1,v_1; \varepsilon), (v'_2,v_2;\varepsilon), (v'_3,v_3;\varepsilon)\}.$$

The resulting graph is shown in Figure 2. A sequence starting at v_1, and containing all the edges in $E_\alpha \cup E_c$ is a checking sequence for the FSM depicted in Figure 1. In order to create a short sequence, the method exposed in [6] adds a new vertex σ and a set of edges $(v_i, \sigma; \varepsilon)$ for all i in $1,\ldots,|V'|$. Finally the edge $(\sigma,v_1;\gamma)$ is also added, where γ represents a "very high cost". An algorithm to find a minimum length tour containing every edge in $\{(\sigma, v_1; \gamma)\} \cup E_\alpha \cup E_c$ is then run. By removing the edge $(\sigma, v_1; \gamma)$ from the tour, a checking sequence is found. In the next section, we show how to use a SAT solver to find such a sequence.

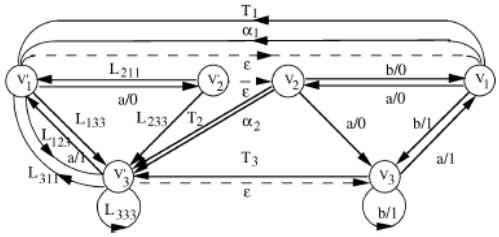

Figure 2: Transformed graph corresponding to Figure 1

III. EXPRESSING THE SEQUENCE AS A BOOLEAN EXPRESSION

We now show step by step how to transform the method described above as a Boolean expression whose resolution would provide us the desired checking sequence. We are going to define Boolean variables capturing the states of the sequence, as well as the constraints to be satisfied. In general, we use an index k ranging from 1 to the size of the path P. The values for $k=i$ can be interpreted as what must be satisfied at the i^{th} node of the path.

The first step is to define the vertices $V=\{v_1, v_2, v_3\}$, and their possible "value" (being at the vertex or not) for every node in the path. We use Boolean variables for this, one for each node and for each k. We thus end up with a set of Boolean variables v_k^i, with $i=1, 2$ and 3 in our case, and k ranging from 1 to the size of the path P. We also need a set of variables σ_k to capture the additional vertex σ. We create the first two formulas, stating that at each k, one and only one of these variables can be true:

$$v_k^1 \vee v_k^2 \vee v_k^3 \vee \sigma_k \qquad (1)$$

$$\neg(v_k^1 \wedge v_k^2) \wedge \neg(v_k^1 \wedge v_k^3) \wedge \neg(v_k^2 \wedge v_k^3) \wedge \neg(v_k^1 \wedge \sigma_k) \wedge \neg(v_k^2 \wedge \sigma_k) \wedge \neg(v_k^3 \wedge \sigma_k) \quad (2)$$

Similarly, we define a set of Boolean variable v'^i_k for vertex $v'_i \in V'$. When these variables are true, so are the v_is:

$$v'^1_k \rightarrow v_k^1 \quad (3)$$
$$v'^2_k \rightarrow v_k^2 \quad (4)$$
$$v'^3_k \rightarrow v_k^3 \quad (5)$$

For convenience, we define a Boolean Nov'_k to state that v_i is true but not v_i':

$$Nov'_k \rightarrow \neg(v'^1_k \vee v'^2_k \vee v'^3_k) \quad (6)$$

We now define the inputs, with a set of variables for each input of the alphabet *a* and *b*, plus the two special inputs ε and γ. The variables are a_k, b_k, ε_k and γ_k. The following formulas specify when these inputs can be used:

$$a_k \rightarrow (v_k^1 \wedge v_{k+1}^2) \vee (v_k^2 \wedge v_{k+1}^3) \vee (v_k^3 \wedge v_{k+1}^1) \quad (7)$$
$$b_k \rightarrow (v_k^1 \wedge v_{k+1}^3) \vee (v_k^2 \wedge v_{k+1}^1) \vee (v_k^3 \wedge v_{k+1}^3) \quad (8)$$
$$\gamma_k \rightarrow \sigma_k \wedge v_{k+1}^1 \wedge Nov'_{k+1} \quad (9)$$
$$\varepsilon_k \rightarrow (v'^1_k \wedge v_{k+1}^1 \wedge \neg v'^1_{k+1}) \vee (v'^2_k \wedge v_{k+1}^2 \wedge \neg v'^2_{k+1}) \vee (v'^3_k \wedge v_{k+1}^3 \wedge \neg v'^3_{k+1}) \vee ((v'^1_k \vee v'^2_k \vee v'^3_k) \wedge \sigma_{k+1}) \quad (10)$$

Again, only one input can be true, and one must be true, at all time:

$$a_k \vee b_k \vee \varepsilon_k \vee \gamma_k \quad (11)$$
$$\neg(a_k \wedge b_k) \wedge \neg(a_k \wedge \varepsilon_k) \wedge \neg(b_k \wedge \varepsilon_k) \wedge \neg(\gamma_k \wedge a_k) \wedge \neg(\gamma_k \wedge b_k) \wedge \neg(\gamma_k \wedge \varepsilon_k) \quad (12)$$

We are now ready to represent the components of the desired checking sequence. The first step is to form the distinguishing sequence $D=ab$. We define a (set of) Boolean variable(s) D_k, indexed by *k*. D_k is true when v_k^i is followed by *D*:

$$D_k \rightarrow a_k \wedge b_{k+1} \quad (13)$$

In order to represent the edges of E_α, E_c, E_T and E'', we are going to track both the starting and ending points of these edges. This is necessary because some of the edges might be included inside other ones and we need to keep them separated. Starting with E_α, we define a set of variables $\alpha_k^{i,start}$ which are true when v_k^i is followed by an α-*sequence* α_i, and another set $\alpha_k^{i,end}$ to track the end of the same α_i. The formulas also capture the fact that E_α is from *V* to *V'*:

$$\alpha_k^{1,start} \rightarrow Nov'_k \wedge v_k^1 \wedge D_k \wedge D_{k+2} \wedge v'^1_{k+4} \quad (14)$$
$$\alpha_k^{2,start} \rightarrow Nov'_k \wedge v_k^2 \wedge D_k \wedge D_{k+2} \wedge D_{k+4} \wedge v'^3_{k+6} \quad (15)$$
$$\alpha_k^{1,start} \Leftrightarrow \alpha_{k+4}^{1,end} \quad (16)$$
$$\alpha_k^{2,start} \Leftrightarrow \alpha_{k+6}^{2,end} \quad (17)$$

Elements of E_c are defined following the same principles:

$$L_k^{123,start} \rightarrow v'^1_k \wedge a_k \wedge D_{k+1} \wedge v'^3_{k+3} \quad (18)$$
$$L_k^{133,start} \rightarrow v'^1_k \wedge b_k \wedge D_{k+1} \wedge v'^3_{k+3} \quad (19)$$
$$L_k^{233,start} \rightarrow v'^2_k \wedge a_k \wedge D_{k+1} \wedge v'^3_{k+3} \quad (20)$$
$$L_k^{211,start} \rightarrow v'^2_k \wedge b_k \wedge D_{k+1} \wedge v'^1_{k+3} \quad (21)$$
$$L_k^{311,start} \rightarrow v'^3_k \wedge a_k \wedge D_{k+1} \wedge v'^1_{k+3} \quad (22)$$
$$L_k^{333,start} \rightarrow v'^3_k \wedge b_k \wedge D_{k+1} \wedge v'^3_{k+3} \quad (23)$$
$$L_k^{123,start} \Leftrightarrow L_{k+3}^{123,end} \quad (24)$$
$$L_k^{133,start} \Leftrightarrow L_{k+3}^{133,end} \quad (25)$$
$$L_k^{233,start} \Leftrightarrow L_{k+3}^{233,end} \quad (26)$$
$$L_k^{211,start} \Leftrightarrow L_{k+3}^{211,end} \quad (27)$$
$$L_k^{311,start} \Leftrightarrow L_{k+3}^{311,end} \quad (28)$$
$$L_k^{333,start} \Leftrightarrow L_{k+3}^{333,end} \quad (29)$$

For E_T and E'', we do not need to track individual instances of the elements of these two sets, so we just create one (set of) variable(s) representing the starting of an element for each state (and the corresponding end of this element):

$$E_{Tk}^{start} \rightarrow Nov'_k \wedge ((v_k^1 \wedge D_k \wedge v'^1_{k+2}) \vee (v_k^2 \wedge D_k \wedge v'^3_{k+2}) \vee (v_k^3 \wedge D_k \wedge v'^3_{k+2})) \quad (30)$$
$$E_{Tk}^{start} \Leftrightarrow E_{Tk+2}^{end} \quad (31)$$
$$E''^{start}_k \rightarrow (v'^3_k \wedge a_k \wedge v'^1_{k+1}) \vee (v'^1_k \wedge a_k \wedge v'^2_{k+1}) \quad (32)$$
$$E''^{start}_k \Leftrightarrow E''^{end}_{k+1} \quad (33)$$

We now have defined all the variables and all the formulas that are required to produce a solution. However, we still need to add some constraints; otherwise the produced solution will not be in accordance with the method of [6]. The first set of additional constraints restricts the edges that can be applied from a vertex in *V'* to elements of E_T, E'' and E_ε:

$$v'^1_k \rightarrow L_k^{123,start} \vee L_k^{133,start} \vee E''^{start}_k \vee \varepsilon_k \quad (34)$$
$$v'^2_k \rightarrow L_k^{233,start} \vee L_k^{211,start} \vee \varepsilon_k \quad (35)$$
$$v'^3_k \rightarrow L_k^{311,start} \vee L_k^{333,start} \vee E''^{start}_k \vee \varepsilon_k \quad (36)$$

Only one of these edges can be followed at any time:

$$\neg(L_k^{123,start} \wedge L_k^{133,start}) \wedge \neg(L_k^{123,start} \wedge E''^{start}_k) \wedge \neg(L_k^{123,start} \wedge \varepsilon_k) \wedge \neg(L_k^{133,start} \wedge E''^{start}_k) \wedge \neg(L_k^{133,start} \wedge \varepsilon_k) \wedge \neg(E''^{start}_k \wedge \varepsilon_k) \quad (37)$$
$$\neg(L_k^{233,start} \wedge L_k^{211,start}) \wedge \neg(L_k^{233,start} \wedge \varepsilon_k) \wedge \neg(L_k^{211,start} \wedge \varepsilon_k) \quad (38)$$

$$\neg(L_k^{311,start} \wedge L_k^{333,start}) \wedge \neg(L_k^{311,start} \wedge E''^{start}_k) \wedge$$
$$\neg(L_k^{311,start} \wedge \varepsilon_k) \wedge \neg(L_k^{333,start} \wedge E''^{start}_k) \wedge$$
$$\neg(L_k^{333,start} \wedge \varepsilon_k) \wedge \neg(E''^{start}_k \wedge \varepsilon_k) \quad (39)$$

Similarly, we need to restrict the edges that can end at a vertex in V' to elements of E_α, E_c, E_T and E'':

$$v'^1_k \rightarrow \alpha_k^{1,end} \vee L_k^{211,end} \vee L_k^{311,end} \vee E_{Tk}^{end} \vee E''^{end}_k \quad (40)$$
$$v'^2_k \rightarrow E''^{end}_k \quad (41)$$
$$v'^3_k \rightarrow \alpha_k^{2,end} \vee L_k^{123,end} \vee L_k^{133,end} \vee L_k^{233,end} \vee L_k^{333,end} \vee E_{Tk}^{end} \quad (42)$$

Only one of these edges can be allowed to end at any time:

$$\neg(\alpha_k^{1,end} \wedge L_k^{211,end}) \wedge \neg(\alpha_k^{1,end} \wedge L_k^{311,end}) \wedge$$
$$\neg(\alpha_k^{1,end} \wedge E_{Tk}^{end}) \wedge \neg(\alpha_k^{1,end} \wedge E''^{end}_k) \wedge$$
$$\neg(L_k^{211,end} \wedge L_k^{311,end}) \wedge \neg(L_k^{211,end} \wedge E_{Tk}^{end}) \wedge$$
$$\neg(L_k^{211,end} \wedge E''^{end}_k) \wedge \neg(L_k^{311,end} \wedge E_{Tk}^{end}) \wedge$$
$$\neg(L_k^{311,end} \wedge E''^{end}_k) \wedge \neg(E_{Tk}^{end} \wedge E''^{end}_k) \wedge \quad (43)$$
$$\neg(\alpha_k^{2,end} \wedge L_k^{123,end}) \wedge \neg(\alpha_k^{2,end} \wedge L_k^{133,end}) \wedge$$
$$\neg(\alpha_k^{2,end} \wedge L_k^{233,end}) \wedge \neg(\alpha_k^{2,end} \wedge L_k^{333,end}) \wedge$$
$$\neg(\alpha_k^{2,end} \wedge E_{Tk}^{end}) \wedge \neg(L_k^{123,end} \wedge L_k^{133,end}) \wedge$$
$$\neg(L_k^{123,end} \wedge L_k^{233,end}) \wedge \neg(L_k^{123,end} \wedge L_k^{333,end}) \wedge$$
$$\neg(L_k^{123,end} \wedge E_{Tk}^{end}) \wedge \neg(L_k^{133,end} \wedge L_k^{233,end}) \wedge$$
$$\neg(L_k^{133,end} \wedge L_k^{333,end}) \wedge \neg(L_k^{133,end} \wedge E_{Tk}^{end}) \wedge$$
$$\neg(L_k^{233,end} \wedge L_k^{333,end}) \wedge \neg(L_k^{233,end} \wedge E_{Tk}^{end}) \wedge$$
$$\neg(L_k^{333,end} \wedge E_{Tk}^{end}) \quad (44)$$

The last formulas are here to ensure that we are going to include the necessary edges in our solution. Firstly, we must ensure that all elements of E_α are included:

$$\bigwedge_{1 \le i \le 2} \left(\bigvee_{0 \le k \le K} \alpha_k^{i,start} \right) \quad (45)$$

Then we must also ensure that all elements of E_c are included:

$$\text{For each } L_{ijk}, \bigvee_{0 \le k \le K} L_k^{xyz,start} \quad (46)$$

Finally, the transition $(\sigma, v_1; \gamma)$ should also be included:

$$\bigvee_{0 \le k \le K} \gamma_k \quad (47)$$

To conclude, the sequence must start and end at v_1. The last value of the index k is the length of the resulting path:

$$v_0^1 = 1, \; Nov'_0 = 1 \quad (48)$$
$$v_K^1 = 1, \; Nov'_K = 1 \quad (49)$$

The set of almost 50 formulas define together a set of constraints that, when satisfied, can be used to infer a checking sequence for the FSM shown Figure 1. The goal will be to use a SAT solver to find a solution, and find a solution that is as short as possible. We describe this approach in the next section.

IV. USING A SAT SOLVER TO FIND THE SEQUENCE

We have implemented the method described here and used a SAT solver to find a solution. Our implementation consists of a Java program that creates the input file to the SAT solver for the input FSM, given the FSM and its α-*sequences*. In order to find the shortest possible solution, we limit the size of the universe that the SAT solver can use. When the universe is too small, no solution is found. This means that a checking sequence of the corresponding size cannot be found for the input FSM when using the method of [6]. By increasing step by step the size of the input universe until a solution is found, we can be sure that the first checking sequence found is the shortest possible (again, for the method [6]. A shorter checking sequence may exist with another method).

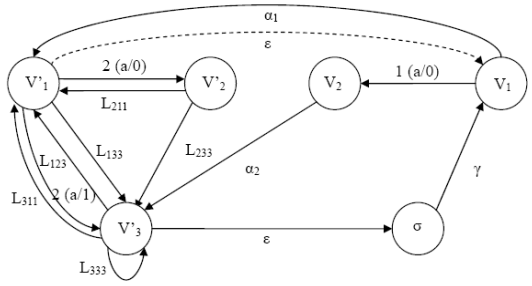

Figure 3: The sub-graph of Figure 2 (augmented with σ) followed by the generated solution.

Our Java program creates a *cnf* file that we input to the SAT solver zChaff [12] in order to find a solution.

With the example described here, the SAT solver fails to find a solution until a size 33 is reached, for which the following solution is found:

$$(v_1, v'_1; \alpha_1)(v'_1, v'_3; L_{133})(v'_3, v'_1; L_{311})(v'_1, v_1; \varepsilon)(v_1, v_2; a/0)(v_2, v'_3; \alpha_2)(v'_3, v'_1; a/1)(v'_1, v'_3; L_{123})$$
$$(v'_3, v'_1; a/1)(v'_1, v'_2; a/0)(v'_2, v'_1; L_{211})(v'_1, v'_2; a/0)(v'_2, v'_3; L_{233})(v'_3, v'_3; L_{333})(v'_3, \sigma; \varepsilon)(\sigma, v_1; \gamma)$$

The resulting tour is depicted on Figure 3. It should be noted that the solution found by our implementation differs from the solution given in [6] for the same example. However, both solutions are of size 33. The discrepancy can easily be explained by the various choices that are made by both algorithms. On the other hand, the fact that we do not find a shorter solution shows that [6]'s solution is optimal. Ours is optimal by construction.

Here is some technical information regarding the experience reported here: we used the version 2007.3.12 of zChaff, using the Conjunction Normal Form. The computer CPU was an Intel Core(TM) Duo Processor 1.66GHz with 1GB of memory, running Microsoft™ Windows XP©, with Cygwin 1.5.24.

Running the program necessitated the creation of 1153 variables, 6377 clauses and 18073 literals. The processing time to find the path of length 33 was 47 milliseconds.

V. Related Work

A similar work is presented in [13], where the authors generate a test sequence by using a SAT solver again. There are several differences between [13] and our work. First, the test sequence generated by [13] is not a checking sequence since they generate test sequence by following the method suggested in [14]. Therefore if an implementation $N \in \Omega(M)$ produces the expected output sequence to the test sequence generated by [13], it is not guaranteed that N would be isomorphic to M, or in other words, it is not guaranteed that N would be a correct implementation of M. On the other hand we always have such a guarantee since we generate a checking sequence. Due to this difference, the results of these two methods cannot be directly compared.

Second difference is the way the test sequence generation is formulated. Without giving too much detail, in [13] the authors also identify some paths that should be included in the test sequence (similar to E_α and E_c in our approach) but then these paths are represented by additional individual nodes (let us call the set of these nodes as R) in an augmentation of the original graph representing the given FSM. A tour going through all the nodes in R is found and the nodes in this tour corresponding to the nodes in R are expanded back to the original paths. The length of the tour found does not correctly reflect the length of the test sequence that will be obtained at the end since the nodes in R may correspond to paths of different lengths although they all contribute as one node to the tour found on the graph.

Third and more importantly, the handling of the overlapping opportunities are different. Although not explicitly addressed in this paper, it is possible to have overlapping between two or more required paths to be included the test sequence. Using these paths in an overlapped manner makes the resultant test sequence shorter. The approach in [13] has to lay down all possible overlapping opportunities and code them in to the augmented graph explicitly, making the graph more complicated.

In our case, the handling of the overlapping opportunities does not require any additional node and it will be possible to encode these opportunities by only tuning Boolean formulae, e.g. by allowing that the paths in E_α and E_c to start and end at the same node.

VI. Conclusion and Future Work

In this study, we were able to reformulate the problem stated in [6] as a set of Boolean variables and functions that can be resolved using a SAT solver, such as zChaff [12] or any such solver. We found the optimal solution without having to implement the actual algorithm.

This approach is promising; we have shown here that the solution works and can be quite effective. We will pursue this work and re-formulate many other methods as Boolean expressions. The goal is to be able to find systematically the optimal solution based on the set of conditions imposed by each method, and regardless of the way the method actually attempts to satisfy the conditions. This will be very useful to evaluate how effectively (or how poorly) currently published methods resolve the constraints that are attempted.

References

[1] Z. Kohavi, *Switching and Finite State Automata Theory*, McGraw-Hill, 1978.

[2] A. Gill, *Introduction to the Theory of Finite-State Machines*, NY: McGraw-Hill, 1962.

[3] F.C. Hennie, "*Fault detecting experiments for sequential circuits", Proc. 5th. Symp*. Switching Circuit Theory and Logical Design, pp.95-110, Princeton, N.J.,1964.

[4] T. Chow, "*Testing software design modeled by finite-state machines*", IEEE Trans. Software Eng., vol.SE-4, pp.178-187, 1978.

[5] G. Gonenc, "*A method for the design of fault detection experiments*", IEEE Trans. On Computer, vol.19, pp.551-558, June 1970.

[6] H. Ural, X. Wu, and F. Zhang, "*On minimizing the length of checking sequence*", IEEE Trans. on Computers, vol.46, pp.93-99, 1997.

[7] R. M. Hierons and H. Ural, "*Reduced length checking sequences*", IEEE Trans. On Computers, vol.51(9), pp.1111-1117, 2002.

[8] J. Chen, R. Hierons, H. Ural, and H. Yenigun, "*Eliminating redundant tests in checking sequences*", Proc. of IFIP TestCom'05, Montreal, Quebec, May 2005, pp.146-158.

[9] R. M. Hierons and H. Ural, "*Optimizing the length of checking sequences*", IEEE Transactions on Computers, Vol.55, No.5, 2006, pp.618-629.

[10] H. Ural and F. Zhang, "*Reducing the Lengths of Checking Sequences by Overlapping*", Proc. of IFIP TestCom'06, New York, May 2006, pp.274-288.

[11] R. M. Hierons, G.-V. Jourdan, H. Ural, and H. Yenigun, "*Using adaptive distinguishing sequences in checking sequence constructions*", Proc. of ACM SAC-SE'08, Fortelaza, Ceara, Brazil, Mar. 2008, pp.682-687.

[12] ZChaff: http://www.princeton.edu/~chaff/zchaff.html.

[13] T. Mori, H. Otsuka, N. Funabiki, A. Nakata, and T. Higashino, "*A test sequence generation method for communication protocols using the SAT algorithm*", Systems and Computers in Japan, vol. 34(11), pp.20-29, 2003.

[14] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar, "*An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours*", IEEE Transactions on Communications, vol. 39(11), pp.1604-1615, 1991.