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  
Sabancı 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, Distinguishing 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 $\Omega$ of $M$ and observing the output sequence produced by $\Omega$ 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 $\Omega$ 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 $\Omega$ 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 $\Omega \in \Omega(M)$, recognize each distinct state in $\Omega$ as a distinct state of $M$ and verify that each transition of $M$ is correctly implemented in $\Omega$. 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 $\Omega$ 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 $\Omega$. 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 $\Omega$ 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 output 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., $\alpha$-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
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 (i.e. a solution to a 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 problem in [6] and it is adopted by some successive checking sequence generation methods [7, 8, 9, 10].

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 CONSTRUCTION METHOD

A. Preliminaries

We represent a deterministic finite state machine (FSM) \( M \) as \((S, s_i, X, \delta, \lambda)\), where \( S \) is a finite set of states with \( n = |S| \), \( s_i \in S \) is the initial state, \( X \) is a finite set of inputs, \( Y \) is a finite set of outputs, \( \delta \) is a state transition function that maps \( S \times X \) to \( S \) and \( \lambda \) 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 \( \text{label}(e) \).

A path \( P = (n_1, n_2; x_1/y_1)(n_2, n_3; x_2/y_2)...(n_{r-1}, n_r; x_{r-1}/y_{r-1}), r > 1 \), 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 \((x_1/y_1)(x_2/y_2)...(x_{r-1}/y_{r-1}) \) is called label of \( P \). \( P \) is represented by \((n_1, n_r; I/O)\), where \( I/O \) is the label of \( P \), \( I = x_1x_2...x_{r-1} \) is called the input portion of \( I/O \), \( O = y_1y_2...y_{r-1} \) 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 \( s_i \) 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., for each \( s_i \in S \), \( s_i \neq s_j; \lambda(s_i,D) \neq \lambda(s_j,D)) \). A distinguishing sequence \( D \) of an FSM \( M \) is then used as follows:

Consider a path \( P \) of \( M \) representing \( M \). Let \( Q = \text{label}(P) \).

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) \((n_p, n_q; T/\lambda(s', T)) \) and \((n_p, n_q; T/\lambda(s', T)) \) are subpaths of \( P \), \( n_p \) and \( n_q \) 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 \((n_l, n_{l+1}; x/y) \) of \( P \) such that \( n_k \) is recognized as \( s_p \) in \( Q \), \( n_{l+1} \) is recognized as \( s_q \) in \( Q \), and \( x/y \) is the label of \( e \).

For the method considered in this paper, if a path of \( G \) representing \( M \) starts from \( s_i \) and verifies all transitions of \( M \), the input portion of this path’s label is a checking sequence of \( M \) [6].


The method proposed by Ural, Wang and Zhang [6] first forms \( \alpha \)-sequences as concatenations of \( D/\lambda(s, D) \) followed by a transfer sequence \( T \) at each state \( s_i \) until the application of the last \( D/\lambda(s, D)T \) is a repetition of an earlier application of \( D/\lambda(s, D)T \) in the path. The \( \alpha \)-sequences are defined in the following way. The first step is to choose subsets \( V_k \) of \( V \) (1 ≤ k
There are two sequences to be the empty sequence for all states. The method first builds a sequence containing all elements of \( D \) and \( \alpha \)-sets. In the case at hand, we have: \( E' = \{(v_j, v_i; \alpha_i), (v_2, v_3; \alpha_3)\} \). The first set of edges, \( E_0 \), is between \( V \) and \( V' \). It connects the nodes of \( V \) to the node of \( V' \) via the \( \alpha \)-sequences. In the case at hand, we have: 

\[
E_0 = \{(v_1, v'_1; \alpha_1), (v_2, v'_2; \alpha_2)\}.
\]

The second set of edges, \( E_0 \), 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: 

\[
E_1 = \{(v'_1, v'_2; L_{12}), (v'_1, v_3; L_{13}), (v'_2, v'_3; L_{23}), (v'_2, v'_3; L_{13}), (v'_3, v'_3; L_{13})\},
\]

where

\[
L_{12} = (aD)/001, L_{13} = (bD)/(111), L_{23} = (aD)/(111),
\]

\[
L_{31} = (bD)/(000), L_{32} = (aD)/(101), L_{33} = (bD)/(111).
\]

The third set of edges, \( E_3 \), is from \( V' \) to \( V' \). It is created by joining elements of \( V \) to elements of \( V' \) by applying \( D \):

\[
E_3 = \{(v_1, v'_1; T_1), (v_2, v'_3; T_3), (v_3, v'_3; T_3)\},
\]

where

\[
T_1 = (D)/(000), T_2 = (D)/(011), T_3 = (D)/(111).
\]

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 to use some flexibility of 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

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

Finally, a fifth set \( E_5 \) is created, joining every vertex of \( V' \) to the corresponding vertex of \( V \) by an edge labeled \( e \):

\[
E_5 = \{(v'_1, v'_2; e), (v'_2, v'_3; e), (v'_3, v'_3; e)\}.
\]

The resulting graph is shown in Figure 2. A sequence starting at \( v_i \), and containing all the edges in \( E_0 \cup E_5 \) 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 \( \alpha \) and a set of edges \( \{v_i, \alpha, e\} \) for every \( i \) ranging from 1 to the size of the path \( P \). Finally the edge \( (\alpha, v'_j; \gamma) \) is added, where \( \gamma \) represents a “very high cost”. An algorithm to find a minimum length tour containing every edge in \( \{(\alpha, v'_j; \gamma)\} \cup E_0 \cup E_5 \) is then run. By removing the edge \( (\alpha, v'_j; \gamma) \) 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 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 \( \alpha \)-sequences, \( \alpha_1(DD)/(0000) \) and \( \alpha_2(DDD)/(011111) \). Note that in this example the transfer sequence \( T_1 \) (following the application of \( D \) at state \( s_1 \)) 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 \( E \):

\[
E_0 = \{(v_1, v'_1; \alpha_1), (v_2, v'_2; \alpha_2)\}.
\]

The second set of edges, \( E_1 \), 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:

\[
E_1 = \{(v'_1, v'_2; L_{12}), (v'_1, v'_3; L_{13}), (v'_2, v'_3; L_{23}), (v'_2, v'_3; L_{13}), (v'_3, v'_3; L_{13})\},
\]

where

\[
L_{12} = (aD)/(001), L_{13} = (bD)/(111), L_{23} = (aD)/(111),
\]

\[
L_{31} = (bD)/(000), L_{32} = (aD)/(101), L_{33} = (bD)/(111).
\]

The third set of edges, \( E_3 \), is from \( V' \) to \( V' \). It is created by joining elements of \( V \) to elements of \( V' \) by applying \( D \):

\[
E_3 = \{(v_1, v'_1; T_1), (v_2, v'_3; T_3), (v_3, v'_3; T_3)\},
\]

where

\[
T_1 = (D)/(000), T_2 = (D)/(011), T_3 = (D)/(111).
\]

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 to use some flexibility of 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

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

Finally, a fifth set \( E_5 \) is created, joining every vertex of \( V' \) to the corresponding vertex of \( V \) by an edge labeled \( e \):

\[
E_5 = \{(v'_1, v'_2; e), (v'_2, v'_3; e), (v'_3, v'_3; e)\}.
\]

The resulting graph is shown in Figure 2. A sequence starting at \( v_i \), and containing all the edges in \( E_0 \cup E_5 \) 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 \( \alpha \) and a set of edges \( \{v_i, \alpha, e\} \) for every \( i \) ranging from 1 to the size of the path \( P \). Finally the edge \( (\alpha, v'_j; \gamma) \) is added, where \( \gamma \) represents a “very high cost”. An algorithm to find a minimum length tour containing every edge in \( \{(\alpha, v'_j; \gamma)\} \cup E_0 \cup E_5 \) is then run. By removing the edge \( (\alpha, v'_j; \gamma) \) 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.
Similarly, we define a set of Boolean variables $v'_k$ for vertex $v_j \in V'$. When these variables are true, so are the $v_j$'s:

$$v'_k \rightarrow v'_i$$  \hspace{1cm} (3)
$$v'_k \rightarrow v'_j$$  \hspace{1cm} (4)
$$v'_k \rightarrow v'_l$$  \hspace{1cm} (5)

For convenience, we define a Boolean $Nov'_k$ to state that $v'_i$ is true but not $v'_j$:

$$Nov'_k \rightarrow -(v'_i \lor v'_j \lor v'_l)$$  \hspace{1cm} (6)

We now define the inputs, with a set of variables for each input of the alphabet $\alpha$ and $\beta$, plus the two special inputs $\epsilon$ and $\gamma$. The variables are $a_i$, $b_i$, $e_i$ and $\gamma_i$. The following formulas specify when these inputs can be used:

$$a_i \rightarrow (v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1})$$  \hspace{1cm} (7)
$$b_i \rightarrow (v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1})$$  \hspace{1cm} (8)
$$\gamma_i \rightarrow \sigma_i \land v'_{i+1} \land Nov'_{i+1}$$  \hspace{1cm} (9)
$$e_i \rightarrow (v'_i \land v'_{i+1}) \lor -(v'_i \land v'_{i+1}) \lor -(v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1}) \lor (v'_i \land v'_{i+1}) \lor \sigma_i$$  \hspace{1cm} (10)

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

$$a_i \lor b_i \lor e_i \lor \gamma_i$$  \hspace{1cm} (11)
$$-(a_i \land b_i) \land -(a_i \land e_i) \land -(b_i \land e_i) \land -(\gamma_i \land e_i)$$  \hspace{1cm} (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) $D_i$, indexed by $k$. $D_i$ is true when $v'_i$ is followed by $D_i$:

$$D_i \rightarrow a_i \land b_{i+1}$$  \hspace{1cm} (13)

In order to represent the edges of $E_{\alpha}$, $E_{\beta}$, $E_{\epsilon}$ and $E_{\gamma}$, 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 $E_{\alpha}$, we define a set of variables $\alpha_{i}^{\text{start}}$ which are true when $v'_i$ is followed by an $\alpha$-sequence $\alpha_i$, and another set $\alpha_{i}^{\text{end}}$ to track the end of the same $\alpha_i$. The formulas also capture the fact that $E_{\alpha}$ is from $V$ to $V'$:

$$\alpha_{i}^{\text{start}} \rightarrow Nov'_i \land v'_i \land D_i \land D_{i+1} \land v'_{i+4}$$  \hspace{1cm} (14)
$$\alpha_{i}^{\text{end}} \rightarrow Nov'_i \land v'_i \land D_i \land D_{i+1} \land D_{i+4} \land v'_{i+6}$$  \hspace{1cm} (15)

Similarly, we define a set of Boolean variable $v''_k$ for vertex $v'_j \in V'$. When these variables are true, so are the $v_j$'s:

$$v''_k \rightarrow v'_i$$  \hspace{1cm} (16)
$$v''_k \rightarrow v'_j$$  \hspace{1cm} (17)

Elements of $E_{\alpha}$ are defined following the same principles:

$$L_{k_{\alpha}}^{\text{start}} \rightarrow v''_k \land a_k \land D_{k+1} \land v'_{k+3}$$  \hspace{1cm} (18)
$$L_{k_{\beta}}^{\text{start}} \rightarrow v''_k \land b_k \land D_{k+1} \land v'_{k+3}$$  \hspace{1cm} (19)
$$L_{k_{\epsilon}}^{\text{start}} \rightarrow v''_k \land a_k \land D_{k+1} \land v'_{k+3}$$  \hspace{1cm} (20)
$$L_{k_{\gamma}}^{\text{start}} \rightarrow v''_k \land b_k \land D_{k+1} \land v'_{k+3}$$  \hspace{21}

$$L_{k_{\alpha}}^{\text{start}} \rightarrow v''_k \land a_k \land D_{k+4} \land v'_{k+3}$$  \hspace{1cm} (21)
$$L_{k_{\beta}}^{\text{start}} \rightarrow v''_k \land b_k \land D_{k+4} \land v'_{k+3}$$  \hspace{1cm} (22)
$$L_{k_{\epsilon}}^{\text{start}} \rightarrow v''_k \land a_k \land D_{k+4} \land v'_{k+3}$$  \hspace{1cm} (23)
$$L_{k_{\gamma}}^{\text{start}} \rightarrow v''_k \land b_k \land D_{k+4} \land v'_{k+3}$$  \hspace{1cm} (24)

$$L_{k_{\alpha}}^{\text{end}} \rightarrow L_{k_{\alpha}}^{\text{end}}$$  \hspace{1cm} (25)
$$L_{k_{\beta}}^{\text{end}} \rightarrow L_{k_{\beta}}^{\text{end}}$$  \hspace{1cm} (26)
$$L_{k_{\epsilon}}^{\text{end}} \rightarrow L_{k_{\epsilon}}^{\text{end}}$$  \hspace{1cm} (27)
$$L_{k_{\gamma}}^{\text{end}} \rightarrow L_{k_{\gamma}}^{\text{end}}$$  \hspace{1cm} (28)
$$L_{k_{\alpha}}^{\text{end}} \rightarrow L_{k_{\alpha}}^{\text{end}}$$  \hspace{1cm} (29)

For $E_{\epsilon}$ and $E_{\gamma}$, 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):

$$E_{k_{\epsilon}}^{\text{start}} \rightarrow Nov'_i \land ((v'_i \land D_i \land v'_{i+1}) \lor (v'_i \land D_i \land v'_{i+1}) \lor (v'_i \land D_i \land v'_{i+2} \lor \sigma_i)$$  \hspace{1cm} (30)

$$E_{k_{\epsilon}}^{\text{end}} \rightarrow E_{k_{\epsilon}}^{\text{end}}$$  \hspace{1cm} (31)

$$E_{k_{\gamma}}^{\text{start}} \rightarrow (v'_i \land a_k \land v'_{i+1}) \lor (v'_i \land a_k \land v'_{i+1})$$  \hspace{1cm} (32)

$$E_{k_{\gamma}}^{\text{end}} \rightarrow E_{k_{\gamma}}^{\text{end}}$$  \hspace{1cm} (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 $E_{\alpha}$, $E_{\beta}$ and $E_{\gamma}$:

$$v'_i \rightarrow L_{k_{\alpha}}^{\text{start}} \lor L_{k_{\beta}}^{\text{start}} \lor L_{k_{\alpha}}^{\text{end}}$$  \hspace{1cm} (34)
$$v'_i \rightarrow L_{k_{\beta}}^{\text{start}} \lor L_{k_{\alpha}}^{\text{end}}$$  \hspace{1cm} (35)
$$v'_i \rightarrow L_{k_{\alpha}}^{\text{start}} \lor L_{k_{\beta}}^{\text{end}}$$  \hspace{1cm} (36)

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

$$-(L_{k_{\alpha}}^{\text{start}} \land L_{k_{\beta}}^{\text{start}}) \land -(L_{k_{\beta}}^{\text{start}} \land L_{k_{\alpha}}^{\text{end}})$$  \hspace{1cm} (37)

$$-(L_{k_{\beta}}^{\text{start}} \land L_{k_{\alpha}}^{\text{end}}) \land -(L_{k_{\alpha}}^{\text{end}} \land L_{k_{\gamma}}^{\text{end}})$$  \hspace{1cm} (38)
Similarly, we need to restrict the edges that can end at a vertex in \( V \) to elements of \( E_0, E_1, E_2 \) and \( E' \):

\[
\begin{align*}
\alpha_k^{\text{end}} & \lor \text{\( F_{k}^{11, \text{end}} \lor F_{k}^{13, \text{end}} \lor E_{k}^{11, \text{end}} \lor E_{k}^{13, \text{end}} \)} \\
\alpha_k^{\text{end}} & \lor \text{\( F_{k}^{11, \text{end}} \lor F_{k}^{13, \text{end}} \lor E_{k}^{11, \text{end}} \lor E_{k}^{13, \text{end}} \)} \\
\alpha_k^{\text{end}} & \lor \text{\( F_{k}^{11, \text{end}} \lor F_{k}^{13, \text{end}} \lor E_{k}^{11, \text{end}} \lor E_{k}^{13, \text{end}} \)} \\
\alpha_k^{\text{end}} & \lor \text{\( F_{k}^{11, \text{end}} \lor F_{k}^{13, \text{end}} \lor E_{k}^{11, \text{end}} \lor E_{k}^{13, \text{end}} \)} \\
\alpha_k^{\text{end}} & \lor \text{\( F_{k}^{11, \text{end}} \lor F_{k}^{13, \text{end}} \lor E_{k}^{11, \text{end}} \lor E_{k}^{13, \text{end}} \)}
\end{align*}
\]

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

\[
\begin{align*}
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}} \\
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}} \\
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}} \\
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}} \\
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}} \\
\alpha_k^{\text{end}} & \land L_{k}^{11, \text{end}}
\end{align*}
\]

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 \( E_0 \) are included:

\[
\big( \lor_{1 \leq i \leq 2} \alpha_i^{\text{start}} \big)
\]

Then we must also ensure that all elements of \( E_1 \) are included:

\[
\lor_{0 \leq i \leq k} Y_i
\]

Finally, the transition \((\sigma, v_i; y)\) should also be included:

\[
\lor_{0 \leq i \leq k} Y_i
\]

To conclude, the sequence must start and end at \( v_0 \). The last value of the index \( k \) is the length of the resulting path:

\[
\begin{align*}
v_0' & = 1, \text{ Nov'}_0 = 1 \\
v_k' & = 1, \text{ Nov'}_k = 1
\end{align*}
\]

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 \( \alpha \)-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 \( \sigma \) followed by the generated solution.](image)

Our Java program creates a \( \text{cnf} \) 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:

\[
(v_1, v_2; \alpha_2, L_{13}; v_2, v_3; L_{23}) (v_1, v_2; \alpha_2, L_{13}; v_2, v_3; L_{23}) (v_1, v_2; \alpha_2, L_{13}; v_2, v_3; L_{23}) (v_1, v_2; \alpha_2, L_{13}; v_2, v_3; L_{23})
\]

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 $E_a$ 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 $E_a$ 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