Transition Coverage Testing for Simulink/Stateflow Models Using Messy Genetic Algorithms

Jungsup Oh
Sungkyunkwan University
300 Chencheon Dong
Suwon, South Korea
jungsup.oh@skku.edu

Mark Harman
University College London
Gower Street, London
WC1E 6BT, UK
m.harman@cs.ucl.ac.uk

Shin Yoo
University College London
Gower Street, London
WC1E 6BT, UK
s.yoo@cs.ucl.ac.uk

ABSTRACT
This paper introduces a messy-GA for transition coverage of Simulink/StateFlow models. We introduce a tool that implements our approach and evaluate it on three benchmark embedded system Simulink models. Our messy-GA is able to achieve statistically significantly better coverage when compared to both random search and to a commercial tool for Simulink/StateFlow model Testing.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging; I.6.4 [Simulation and Modeling]: Model Validation and Analysis

General Terms
Algorithms

Keywords
Search-Based Software Engineering, Model-Based Testing

1. INTRODUCTION
This paper is concerned with search based testing for transition coverage of Simulink/Stateflow models. State based models are widely used in the design and implementation of embedded systems. This is an important domain for search based testing because of the prevalence of embedded systems. It has been estimated that the total number of embedded systems is 6.8 systems. It has been estimated that the total number of embedded systems is 6.8 systems. This is an important domain for search based testing because of the prevalence of embedded systems. Our messy-GA is able to achieve statistically significantly better coverage when compared to both random search and to a commercial tool for Simulink/StateFlow model Testing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

than 300 papers on Search Based Software Testing (SBST) only about 5% concern test data generation for state based models and fewer than 2% concern Simulink models.

Simulink/Stateflow (SL/SF) is one of the most popular modelling languages in the automotive and aerospace domain. Simulink [34] is a software package for modelling, simulation and analysis of system-level design features of dynamic systems. A Simulink model consists of connected blocks, each of which is a functional unit. Models can be designed hierarchically using a block as a subsystem of the other block. Simulink also includes Stateflow [36] blocks, which enable modelling of event-based functionalities.

Figure 1 contains a SL/SF model for the automotive power-window system (PW) used in our evaluation in Section 4. Simulink blocks and a Stateflow block are described in detail on the right hand side of the figure. It contains two of the features that illustrate the challenges involved in testing SL/SF models.

The first challenge is the operational semantics of SL blocks, such as the Discrete-Time Integrator block in Figure 1. The integrator block is often used to integrate or accumulate signals. Some transition triggers may require this block to accumulate a specific amount of signal before they become activated. The existence of such blocks significantly reduces the chance of generating a valid test without a guide.

The second challenge is the existence of cyclic paths. In the model, the trigger for transition (17) requires hit to be 0. However, the only way of reaching the source state (6) is to go over the cyclic paths between state (3), (5) and (6) twice: once using transition (14) and the second time using transition (16). The shortest transition tour that covers transition (17) would be either (11)-(12)-(14)-(15)-(12)-(10)-(16)-(17) or (11)-(12)-(10)-(16)-(15)-(12)-(14)-(17).

This paper introduces an SBST approach for SL/SF models using messy-GA (mGA). Naturally the choice of representation and crossover operator are important success drivers for any evolutionary approach [16]. Our approach can generate test data for transition coverage and assumes no, a priori, knowledge of the length of the input sequence. It can handle Simulink blocks, concurrency, nested loops and cyclic paths and it is evaluated on three practical SL/SF models: a stop watch model and two automotive system models. We use a modified cut and splice crossover operator that promotes higher diversity of input sequence length,

Figure 1: SL/SF model of a car power-window. It contains two factors that are challenging to automated test data generation. First, the Discrete-Time Integrator SF block requires a certain number of accumulated input to generate a signal. Second, the SF block contains a cyclic path.

thereby guiding the search towards those paths that either are cyclic or include accumulation blocks.

The primary contributions of this paper are as follows:

1. The paper introduces an automated test generation approach for SL/SF models using messy-GA. The messy-GA allows us to generate transition coverage adequate input sequences without fixing the length of sequences in advance. The approach is also able to cope with not only cyclic paths and concurrency but also Simulink blocks such as integrator, delay and change detection.

2. The paper introduces a tool that implements our approach and presents an empirical study to evaluate it. The results of the study reveal that our approach outperforms both the random approach and a commercially available tool with respect to transition coverage criterion. For each of the three systems the performance of our approach is significantly better than both random search and the commercial tool at the 99% confidence level (for both t-test and the non parametric Wilcoxon test).

The organisation of the remainder of the paper is as follows. Section 2 formalises the definitions of SL/SF models upon which our approach is based, while Section 3 describes our algorithm and its implementation. Section 4 describes the empirical evaluation of our approach, the results of which are discussed in Section 5. In order to give the reader the context in which our work resides, Section 6 describes the previous work to which our work is related and sets out some directions for future work that draw in our results as well as this previous work. Section 7 concludes.

2. DEFINITIONS

There are many variations of SL/SF models with subtle differences. For clarity, this section defines the SL/SF models used in or work. A state-based model is defined as a tuple $M = (S, \Pi, V, T)$, where $S$ is a set of states, $\Pi$ is a set of events, $V$ is a set of variables and $T$ is a set of transitions [20]. Let $\Theta$ denote an interpretation of $V$ that assigns an initial value to each variable $v$ in $V$. For example, the initial value of the variable $m$ in Figure 2 is defined by $\Theta(m) = 0$. A transition $t \in T$ is a tuple $(s_s, t_r, g_r, a, s_t)$: $s_s \in S$ is the source state of the transition, $s_t \in S$ is the target state of the transition, $t_r$ is a predicate on $\Pi$. $g_r$ is a predicate on $V$ and $a$ is a set of assignments to $V$. Let $s_S(t)$ denote the source state of a specific transition $t$, $s_T(t)$ the target state of a specific transition $t$ and so on.

We partition the set of variables, $V$, into three disjoint subsets: $V_I$, $V_O$ and $V_L$, each representing a set of input, output and local variables respectively. Input variables are set by an external input of the state-based model. Output variables communicate the output from the state-based model to the external world. Finally, local variables are only used internally.

A Stateflow model can contain a hierarchical and/or a concurrent structure defined over states. A state is either basic or composite. A basic state is one without any child state, whereas a composite state contains at least one child state (a set of which is denoted by $ch(s)$). For example, State 1 in Figure 2 is a composition state, while all others are basic. A composite state $s$ is classified either as an OR-state ($s_I$) or an AND-state ($s_k$). An OR-state has only one active sub-state at any time, i.e., the model in Figure 2 being in State 1 implies that the model is actually in either State 2, 3 or 4 but not in more than one of the sub-states at any time. On the other hand, child states of an AND-state are all active simultaneously. Let $s_I$ and $s_k$ represent the set of OR- and AND-states respectively; it follows that $S = S_I \cup S_k$. Finally, any Stateflow model contains a unique state called the root state, which is the state at the highest level of this hierarchy. In Figure 2, State 1 is the root state.

A configuration $C$ is a maximal set of states in which a system can be simultaneously. More formally, $C \subseteq S$ is called a configuration if all the following conditions are met:

- $C$ contains the root state of $M$
- $\forall s_k \in S, (s_k \cup ch(s_k) \in C) \vee (s_k \cup ch(s_k) \notin C)$
- $\forall s_I \in S, (s_I \cap C \cap ch(s_I) \cap C = 1) \vee (s_k \cup ch(s_k) \notin C)$

Therefore, each configuration can be uniquely characterised by its basic states. There are 4 configurations in Figure 2,
which are \(\{1, 2\}, \{1, 3\}, \{1, 4\}\) and \(\{1, 5\}\). Since the root state is always included in any configuration, these notations can be simplified to \(\{2\}, \{3\}, \{4\}\) and \(\{5\}\).

We define a base node to be a snapshot of an executable SL/SF model. More formally, a base node \(B\) is a tuple \((C, V_{gr(t_0(C))}, i_c)\) given a configuration \(C\). First, let \(t_0(C)\) be the set of transitions whose source state is in the configuration \((s_S(t) \in C)\). The set \(gr(t_0(C))\) contains the source state in the configuration. The algorithm saves the collateral coverage \(gr\) that represents the configuration \(C\). Two distinct base nodes may have the same configuration \(C\) but contain different transition triggers, \(V_{gr(t_0(C))}\). The formal definition of the base node forms the phenotype representation, which is described in more detail in Section 3. The last element of the base node tuple \(B\) is an input sequence, \(i_c\). This is a sequence of inputs that will lead the SL/SF model from its initial state to the state of the snapshot denoted by the pair of configuration \(C\) and the transition triggers \(V_{gr(t_0(C))}\).

3. TEST DATA GENERATION ALGORITHM

The test case generator aims to find test cases that achieve the given coverage criterion. This paper uses all transition coverage. Algorithm 1 shows the pseudo-code for the main algorithm for transition coverage test data generation. The algorithm first sorts all transitions in the given SL/SF model by topological order. For example, with the model depicted in Figure 2, the transition from State 3 to State 4 should be attempted after the test case for transition from State 2 to State 3 has been found, providing the base node that contains the source state in the configuration. The algorithm starts with an initial base node \(b_0\) that represents the initial state of the model. In the main loop, the algorithm searches for test case for each transition by invoking \texttt{FindTestCase}. The algorithm saves the collateral coverage achieved for non-target transitions. The \texttt{FindTestCase} procedure is essentially a messy Genetic Algorithm applied to test case generation for SL/SF models.

**Algorithm 1: Main Algorithm for Test Data Generation for Transition Coverage**

**Input:** A SL/SF model \(M = (S, \Pi, V, T)\)

**Output:** A list of base-nodes, \(L\), that satisfies the transition coverage criterion

\[
\begin{align*}
(1) & \quad \text{Sort } T \text{ in topological order} \\
(2) & \quad L \leftarrow \emptyset \\
(3) & \quad \text{Add the initial base node, } b_0, \text{ to } L \\
(4) & \quad \text{foreach } t \in T \\
(5) & \quad L \leftarrow L \cup \text{FindTestCase}(t) \\
(6) & \quad \text{if transition coverage is satisfied then break} \\
(7) & \quad \text{return } L
\end{align*}
\]

**Fitness Evaluation:** \texttt{FindTestCase} uses Korel’s objective function table [27] to calculate the branch distance for each predicate that forms the guard of the transition of interest. Branch distance is the sum of objective function values of each term in the target transition; it measures how close the test input is to satisfying the guard of the target transition. Since the input sequence has an unspecified and potentially unbounded number of steps, it is possible for an individual to visit the target transition multiple times, initiating the calculation of branch distance multiple times. However, the algorithm uses only the final measurement of branch distance. Despite the relative simplicity of this approach, our results indicate that it can be effective for SL/SF models, though undoubtedly it would perform poorly if used for SBST of programs rather than models. A further fortunate side effect of our approach is that the unbounded nature of the input means that there is no need for normalising fitness values, avoiding issue with normalisation[4].

**Genetic Operators:** We use binary tournament selection and a mutation operator adds a single step to the input sequence by adding random input values. These are entirely standard. However, we use a cross over operator and representation that is specifically designed for the SL/SF problem in order to overcome the challenges for SL/SF testing mentioned in the introduction. With test data generation for SL/SF models, a key insight is that crossover should be constrained to occur only at specific points in order to preserve building blocks and to yield recombinations of compatible genetic material, rather than arbitrary material.

We employ a ‘cut and splice’ crossover that allow us to cut two parents at different locations and splice the parts together. This naturally entails a variable length chromo-

![Figure 2: A simple Stateflow model](image.png)
some representation, which is why we combine this crossover with a messy-GA algorithm. Our goal is to ensure that sequences are spliced at points which share the same configuration. Where this is possible for two parents, a cut and splice point is chosen so that the input flows continuously from the part of the child chromosome taken from one parent to that part of the child taken from the other parent. Where this is not possible the crossover chooses an arbitrary splice point.

**Implementation:** Figure 3 describes the architecture of the test data generator, which contains 3 main components: executable model generator, coverage goal generator and test case generator. The executable model generator creates an executable model from an SL/SF model in order to dynamically evaluate test inputs. The coverage goal generator provides the list of goal elements of the SL/SF model under test to be covered by test cases following a coverage criterion. The test case generator uses the messy-GA to construct test cases for the criteria supplied to it by the coverage goal generator on the model from the executable model generator.

A feedback report is generated for a guard contains the following information: 1) id of the guard, 2) boolean evaluation and branch distance of each term in the guard (following Trace et al.) and 3) the final boolean evaluation of the guard. An example report is T1(T(-3.0)F(2.0)T. This is interpreted as follows: the guard for transition T1, which consists of two terms, has been evaluated; the first term has been evaluated to be True with the branch distance of -3.0 while the second term has been evaluated to be False with the branch distance of 2.0. The guard itself has been resolved to be True, thereby initiating the transition.

**4. EXPERIMENTAL SETUP**

The empirical study uses 3 SL/SF models from the MATLAB benchmark suite in order to evaluate the proposed approach with respect to the full transition coverage criterion. All 3 models have been studied for test data generation in the literature. The PW model contains a relatively simple Stateflow chart but a complicated Simulink block (an integrator). It also contains a cyclic path. The sf_car model represents an automatic transmission controller of a modern car; it contains concurrent states and events. The logic determines when the gear goes up or down. The stopwatch model is a model of a stopwatch logic. This model has several junctions, some of which work like nested loops. This makes stopwatch model an ideal subject for checking whether a test data generation algorithm can solve the loop problem. One transition in this model requires 6,000 (= 60 * 100) iterations of the loop before being triggered. It also contains several Simulink blocks.

The proposed messy-GA approach is compared to two other test data generation approaches: random test data generation and a well-known commercial tool for Model-Based Testing called Reactis. Random approach provides a baseline and a sanity check. The internals of Reactis tool is not known as it is a commercially available tool but it is said to use guided simulation of the model. The population size of the messy-GA has been set to 100, while the mutation rate has been set to 0.1. In order to cater for the inherent randomness in the algorithm, each of the 3 algorithms have been repeated 100 times.

The research questions for the study are now defined. **RQ1** concerns the effectiveness of the use of messy-GA in order to generate test data for SL/SF models while **RQ2** asks a more qualitative question and is answered by analysing the SL/SF models as well as the results of the empirical study. **RQ1. Effectiveness:** for a given coverage criterion, how does messy-GA perform against other test data generation approaches? Section 5 answers **RQ1** by observing the coverage values achieved by different approaches. On the contrary, **RQ2. Insights:** are there any specific properties of SL/SF models that make messy-GA an effective/ineffective tool for test data generation? Section 5.1 answers **RQ2** by analysing the test input generated by the messy-GA in detail.

**5. RESULTS**

Table 2 shows the results of our experiments, while Figure 4 depicts the mean coverage observed from 100 runs of each tool for each subject model, along with the 95% confidence level intervals. The column labelled ‘Success Rate’ in Table 2 denotes the percentage of successful runs; those producing 100% transition coverage. For all 3 subject models the messy-GA produces not only higher mean coverage values but also higher success rates. Reactis fails to generate any test data for sf_car because it cannot cope with some complicated Simulink blocks included in the model. In
other models the messy-GA outperforms both the random tool and Reactis.

To gain more confidence, the results in Table 2 have been statistically tested using a one-sided t-test (See Table 3). A one-sided Wilcoxon non-parametric test was also applied (with identical significance at the 99% level) indicating that no assumptions need be made about the distributions of results. The null hypothesis is that the transition coverage achieved by the messy-GA is equal to coverage values achieved by the other two approaches. The alternative hypothesis is that messy-GA achieves higher coverage than the other two. For all cases, the null hypothesis is rejected with confidence, i.e. messy-GA does outperform other two approaches with statistical significance. This answers RQ1: the messy-GA does produce higher transitional coverage compared to both random and a commercial tool.

5.1 Insight Result for PW model

Let us revisit the PW model example from the introduction to examine how our messy-GA generates test input. Table 4 gives an input sequence generated by the messy-GA for transition (17) of the model PW. For brevity we list only salient parts of the entire input sequence; the shortest input sequence generated contained more than 200 steps. Each row contains a single step. The input sampling rate is 0.05 seconds, hence the time column.

It can be seen that during the part (a) of the input sequence, the integrator reached the pre-defined accumulation level and triggered the \texttt{winhit} signal, which resulted in transition (16) being activated in the next step. Similarly, the part (b) led to triggering of \texttt{winhit} again. However, it is transition (14) that is triggered after (b), leading the execution towards the subsequent execution of transition (17).

Reactis was unable to cover \texttt{sf\_car} so no test was possible in this case.

This illustrates how the messy-GA increases the length of an input sequence until the integrator block triggers the necessary event. The algorithm also guides the execution so that, when the second time the integrator triggers the necessary event, the correct cyclic path is chosen in order to cover a specific transition. This shows how messy-GA can overcome the some of the SL/SF challenges answering RQ2.

6. RELATED AND FUTURE WORK

During last decade, evolutionary algorithm has been widely used to generate test data. It has been successfully applied to many testing problems including path-based testing, mutation testing, stress testing, regression testing as well as testing of Object–Oriented, Aspect–Oriented, concurrent and Agent–Oriented systems. There are several excellent surveys of this previous SBST work [1, 14, 19, 37]. However, as these surveys all reveal, very little of the literature on SBST is concerned with state based models, and an even smaller proportion with SL/SF Models. In this section we review related work on SBST for state based models and its relationship to our work.

There has been considerable work in the SBST for the generation of Unique Input Output sequences and related test sequences of FSM testing [7, 12, 13]. Derderian et al. also applied GA to generate test data for Finite State Machine(FSM) with temporal constraints [8]. The fitness function was based on the number of temporal constraint violations committed by each candidate input sequence. However, SL/SF models are considerably more demanding, since they are extended FSMs and so these FSM testing approaches do not directly apply to SL/SF testing.

Lepticaru et al. [28, 30, 31, 29] presented an application of GA to generate test input that executes a specific path in an extended FSM, drawn from UML models. This work is closer to that required for testing SL/SF, since the FSMs are extended, though the authors do not report on approaches to handle concurrency nor does their formalism include state-flow blocks.

Windisch et al. [47, 48, 49] used Simulated Annealing (SA), Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) in order to generate continuous input signal for real-time SL/SF models: the signals were generated by a sequence of individual signal blocks. Lindlar [32], incorporated Lehmann and Bringmann’s Time Partition Testing process into SBST for models. However, unlike our approach this work retains the concept of an approach level metric as

![Figure 4: Mean coverage over 100 runs](image)

### Table 4: A test input generated by messy-GA for transition (17) of model PW depicted in Figure 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Test Input</th>
<th>Feedbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0.05</td>
<td>1 12</td>
<td>T(0.00)</td>
</tr>
<tr>
<td>0.10</td>
<td>1 14</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>0.15</td>
<td>0 14</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>0.20</td>
<td>0 9</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>1.25</td>
<td>0 9</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>1.30</td>
<td>0 9</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>1.35</td>
<td>0 17</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>4.95</td>
<td>1 14</td>
<td>T(0.00)</td>
</tr>
<tr>
<td>5.00</td>
<td>2 17</td>
<td>T(0.00)</td>
</tr>
<tr>
<td>5.05</td>
<td>1 18</td>
<td>F(1.00)</td>
</tr>
<tr>
<td>5.10</td>
<td>2 18</td>
<td>T(0.00)</td>
</tr>
</tbody>
</table>

Table 3: One Sided t-Test Outcomes

<table>
<thead>
<tr>
<th>Comparison</th>
<th>PW</th>
<th>sf_car</th>
<th>stopwatch</th>
<th>Random</th>
<th>Reactis</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs. Random</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
<td>&lt; 2.2e-16</td>
<td>0.002102</td>
<td>- 1.049e-13</td>
</tr>
</tbody>
</table>
part of fitness and fixed length input sequences. Ghani et al. [10] used GA and SA for switch block path coverage of Simulink model (but, unlike our work, without Stateflow blocks). They report that GA and SA achieved similar coverage, but GA was more often successful, which was one of our motivations for using a GA in our work, which can be thought of as an extension of the work of Ghani et al.

Zhan and Clark [53] combined random testing and search-based test data generation in order to generate branch adequate test data more efficiently. After applying random test data generation, remaining test requirements were targeted using search-based techniques. Zhan and Clark [51] also defined mutation operators for Matlab/Simulink models and use SBST to find mutation adequate test data for these sets of mutants. Zhan and Clark [50] also used a simulation-based approach to SBST for Matlab/Simulink models with basic blocks. They simulate the execution of the block in a ‘black box’ style to attempt to generate test data for the block. They subsequently developed this approach to incorporate elements of symbolic execution to address the state variable problem [52].

Our work also differs from previous work in its method of handling concurrency, which presents further challenge for SBST. Kim et al. [26] use a sequentialisation technique to convert a (concurrent) model into a sequential equivalent. However, this can create a state explosion, as sequentialisation is well-known to suffer from this problem. Katayama et al. [25] seek to overcome any potential state explosion by using an extra graph to denote concurrent interactions, while Ambrosio et al. [2] generate test data for each sequential EFSM before attempting to combine the results for the concurrent ensemble. Our approach avoids the need for sequentialisation, external graphs or piecewise composition because it uses a crossover operation specifically tailored for SL/SF. However, the careful construction of representation and crossover operators for SBSE has been previously studied for other problems such as modularisation [16] and SBST for Object Oriented Programs [5].

Other authors [41, 39, 40, 43] have also considered the problem of state variables in programs. These programs are not state based models, but their use of state variables means that the number of times a method is executed may affect whether a branch is covered, rather than merely the values passed to the method on a single call. This complicates SBST for programs. At the model level, the presence of counter variables creates similar problem. For instance, it may lead to infeasible paths.

Kalaji et al. [22, 24] used GA to guide the search for feasible transition paths (rather than test cases to exercise them). They show that a GA can overcome problems with counter variables. Kalaji et al. [23] also show how the problem of state variables can be overcome using a testability transformation [17]. Complementary to this, Zhao et al. proposed an approach to generate test data for feasible EFSM paths [54]. Kalaji et al.’s work on feasible paths in EFSMs could be combined with a Species Per Path approach [38] to locate sets of feasible paths to those difficult target transitions that transition coverage approaches (like ours) may be unable to cover.

There are other work on generating test data for SL/SF models without using a search-based approach. Satpathy et al. tried to outperform commercial tools by combining different existing approaches [44]: Directed Automated Ran-

dom Testing (DART) [11], hybrid concolic testing [33] and feedback-directed random testing [42] have been applied in conjunction with each other based on a heuristic.

Commercial tools are available for testing SL/SF models. T-VEC [46] from T-VEC technologies automatically generates test cases using domain testing theory. Safety Test Builder [6] from Greensoft generates numerical test cases by building exhaustive execution trees from automata-based specifications. This tool is based on symbolic execution and constraint solving. BEACON Tester [21] from Applied Dynamics International is a tool for generation of code from Simulink models and automatic generation of test vectors. The Automatic Unit Test Tool (AUTT)-part of BEACON creates test vectors. These test vectors target several coverage criteria and other common error sources such as numerical overflows. Simulink Design Verifier [35] from Mathworks generates random test inputs after statically analysing the SL/SF model. Reactis [45] from Reactis Systems uses guided simulations using algorithms and heuristics. It is one of the most famous commercial tools for generating test cases from an SL/SF model.

Our work is the first to present results for transition coverage of Simulink models with Stateflow blocks. It is also, to the author’s knowledge, the first paper to present empirical results that compare SBST approaches with commercial off-the-shelf tools for state based model testing. Therefore, despite the relative lack of work on SL/SF testing compared to other topics in SBST, it is perhaps an encouraging sign of the increasing maturing of the field that it is possible to compare results from prototype SBST tools such as ours to commercial products in this way. Naturally the usual caveats about threats to validity and the degree to which one can generalise from these initial results still apply.

Though our results are encouraging, there remains more to be done. In future work, we shall explore the degree to which recent results [3] on dependence analysis for state based models can help to determine those inputs that can affect whether a transition is covered. This information will be used to investigate whether domain reduction techniques, found successful in search based testing for imperative [15] and aspect oriented [18] programming styles can also be extended to reduce effort and improve effectiveness for search based testing of SL/SF models. Future work will also include a wider evaluation of the proposed approach and exploration of the performance of other search based algorithms such as the hill climbing/Alternating Variable method, Simulated Annealing and Genetic Programming.

7. CONCLUSION

This paper presents a messy-GA based framework for generating test data for SL/SF models. It consists of three major components: the executable model generator, the coverage goal generator and the test case generator. The framework creates an executable model from a SL/SF model (executable model generator), sets up a test plan based on a coverage criterion (coverage goal generator) then dynamically generates test data by applying messy-GA using the feedback from the executable model (test data generator).

The empirical evaluation compared the proposed framework to both the random approach and a commercially available test data generation tool, using 3 widely-studied benchmark SL/SF models. The results show that, without any a-priori knowledge of the required length of test input sequences, messy-GA outperforms other algorithms with re-
spect to the full transition coverage criterion, even when there exist memory-contained Simulink blocks, nested loops or cyclic paths. The proposed framework also consistently outperformed the commercially available tool, which failed to generate any test data for one of the subject models.

Acknowledgement This work was supported by the National Research Foundation of Korea funded by the Korean Government (NRF-2009-352-D00266) and PRCP through NRF of Korea funded by MEST (2010-0020210).

8. REFERENCES


