Static Preprocessing for Automated Structural Testing of Simulink Models

Benjamin Wilmes
Berlin Institute of Technology
Daimler Center for Automotive IT Innovations (DCAITI)
Berlin, Germany
E-Mail: benjamin.wilmes@dcaiti.com

***Abstract*—A feasible automation of testing software models would be of great benefit to industry, given the advantages of early testing as part of an efficient quality assurance process. Despite search-based testing having been applied with promising results to automate structural test data generation for Simulink models, the approach lacks efficiency. This paper features three static-analysis-based preprocessing techniques which are carried out prior to an automated test data search, to mitigate this efficiency problem. The first technique identifies unsatisfiable coverage goals by analyzing the ranges of model internal signals and excludes them from the search. The second preprocessing technique aims at reducing the search space by analyzing which model inputs actually require stimulation in order to reach a certain model state. A third technique sequences the coverage-goal-related search processes in order to maximize collateral coverage and reduce the size of the generated test suite. These additional techniques are able to make the search-based approach considerably more efficient, as results of a case study with our search-based testing tool TASMO, applied to industrial Simulink models, reveal.**

***Keywords*-Search-Based Testing, Static Preprocessing, Automotive Industry, Simulink.**

I. INTRODUCTION

First of all, note that this paper is an extended version of a previous publication [1]. It contains further details on the topic, additional illustrations and an extended case study.

In many of today's application areas, the creation of embedded controller software relies on model-based design paradigms. Various industries, such as the automotive industry, use Matlab Simulink (SL) [2] as the standard tool to create and simulate dynamic models along with a code generator, for instance TargetLink (TL) [3], in order to automatically derive software code from such models.

As they are normally the first executable artifacts within software development processes, SL models play an important role in testing theory. Industrial testing practice, however, usually focuses on higher-level development artifacts, like testing integrated software or systems as a whole. This discrepancy has both traditional and practical reasons. On one hand, current testing processes still require further adaptation to the model-based paradigm. On the other hand, companies are pressed for time in product development and must deal with an increasing demand for innovation. This can lead to a disregard for low-level tests and model tests in particular. Yet focusing too one-sided on tests of higher-level software or system artifacts poses the risk that faults may be found late in the process, which can lead to increased costs, that some faults can hardly be discovered on higher levels, or that certain functionality is not tested at all.

Thus, automating the testing of software models is highly desirable in industrial practice, particularly with regard to what is normally the most time-consuming testing activity: the selection of adequate test cases in the form of model input values (test data). Search-based testing is a dynamic approach to automating this task. It transforms the test data finding problem into an optimization problem and utilizes meta-heuristic search techniques like evolutionary algorithms to solve it. Search-based testing [4] has been studied widely in the past and has also been applied successfully for testing industrial-sized software systems [5]. Both structural (white-box) and functional (black-box) testing can be automated with the search-based approach.

Zhan and Clark [6], as well as Windisch [7], applied search-based testing to structural test data generation for SL models. The work of Windisch not only supports Stateflow (SF) diagrams (which are used fairly often in SL models), but also makes use of an advanced signal generation approach in order to generate realistic test data. While his approach has led to promising results in general, outperforming commercial tools in terms of effectiveness, it lacks efficiency when applied to larger models. Furthermore, it shows difficulties targeting Boolean states and tackling complex dependencies within models [8].

The work presented in this paper is a first step toward overcoming some of these shortcomings by exploiting static model analysis techniques before the search process actually starts. These techniques will therefore be referred to as static preprocessing techniques. Our scope is test data generation for TL-compliant Simulink models. Our primary aim is to improve the efficiency of the approach by Windisch. While our work is targeted at SL models, we believe that the presented concepts are generally portable to similar data-flow diagram types and, at least conceptually, even to structural testing of code.

This paper is structured as follows: Section II introduces search-based testing and its application to structural testing of SL models. Section III presents our approach to supporting the search-based technique by integrating three

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

preprocessing techniques. In detail, we present a signal range analysis (Section III-A) which captures range information of internal model signals and, in this way, allows partial detection of unreachable model states. We then propose a signal dependency analysis for the purpose of search space reduction (Section III-B). Our third contribution is a sequencing approach which derives an order in which coverage goals of a structural test are processed by the search (Section III-C). Insight into our tool prototype and a case study are provided in Section IV and V. An overview of related work in the field of structural test data generation for SL models is provided in Section VI, followed by our conclusions in Section VII.

II. Background

A. Search-Based Structural Testing

Initiated in the 1970s by Miller and Spooner [9] and revived by Korel in the 1990s [10], search-based testing [4] and its application to industrial cases has been extensively studied in the last decade.

The general idea of the search-based approach is pretty simple: a test data finding problem (which surely differs in its nature depending on the kind of testing) is transformed into an optimization problem by defining a cost function, called fitness function. This function rates any test data generated by the deployed search algorithm - usually based on information gained from executing the test object with it. The rating must express, in as much detail as possible, how far the test data is from being the desired test data. An iteratively working search algorithm uses these fitness ratings to distinguish good test data from bad, and based on this, generates new test data in each iterative cycle. This fully automated procedure continues until test data satisfying the search goal(s) has been found, that is, if a fitness rating has reached a certain threshold or until a predefined number of algorithm iterations have been performed. Various search algorithms have been used in the past. Due to their strength in handling diverse search spaces, evolutionary algorithms, like genetic algorithms, were often preferred [11].

Applied to functional testing, the search-based approach is generally utilized to search for violations of a requirement. In this case, a sophisticated fitness function needs to be designed manually when following the standard approach. However, when applied to structural test data generation, fitness functions can be derived completely automatically from the inner structure of the program to be tested.

Structural testing is commonly aimed at deriving test data based on the internal structural elements of the test object, e.g., creating a set of test data which executes all statements of a code function, or all paths in the corresponding control flow graph. Industrial standards like ISO 26262 even demand the consideration of coverage metrics when performing low-level tests. Search-based testing can automate this task for various coverage criteria (like branch or condition coverage) by treating each structural element requiring coverage as a separate search goal, called a coverage goal (CG/CGs in plural). Each CG is accompanied by a specific fitness function. Wegener et al. [12] recommend composing the fitness function of the following two metrics: approach level (positive integer value) and branch distance (real value from 0 to 1). Given a test data's execution path in the control flow graph of the test object's code, the approach level describes the smallest number of branch nodes between the structural element to be covered and any covered path element. To create a more detailed and differing rating of generated test cases, the branch distance reflects how far the test object's execution has been from taking the opposite decision at the covered branch node, which is the closest to the structural element to be covered. This approach is suitable for structural testing of program code, like C or Java code.

B. Application to Dynamic Systems

As model-based development is now established in the automotive industry and practitioners have noticed opportunities to test earlier, Windisch [7] as well as Zhan and Clark [6] have transferred the idea of search-based structural testing from code to model level. For SL models, structural coverage criteria similar to the ones known from code testing exist and are commonly accepted in practice. Before addressing the challenges of applying search-based test data generation to SL models, we give a brief introduction to SL. SL is a graphical data-flow language for specifying the behavior of dynamic systems. Syntactically, a SL model consists of functional blocks and lines connecting them, while most of the blocks are equipped with one or more input ports as well as output ports. The semantics of such a model results from the composed functionalities of the involved block types, e.g., sum blocks, relational blocks or delay functions. In addition, event-driven or state-based functionalities can be realized within SL models using SF blocks. A SF block contains an editable Statechart-like automaton.

When applying search-based structural testing to SL models, two fundamental differences compared to its application on code level arise. First, SL models describe time and state dependent processes. Inputs and outputs of SL models, as well as block-connecting lines, are in fact signals. In order to enable reaching all system states, an execution with input sequences (signals) instead of single input values is required. Such complex test data can only be generated with common search algorithms by compressing the data structure, as done by Windisch [13]. His segment-based signal generation approach also considers the necessity for being able to specify the test data signals to be generated (e.g., amplitude bounds and signal characteristic, like wave or impulse form). Second, the aforementioned fitness function approach cannot be fully adopted since SL models are data flow-oriented. There are no execution paths because

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

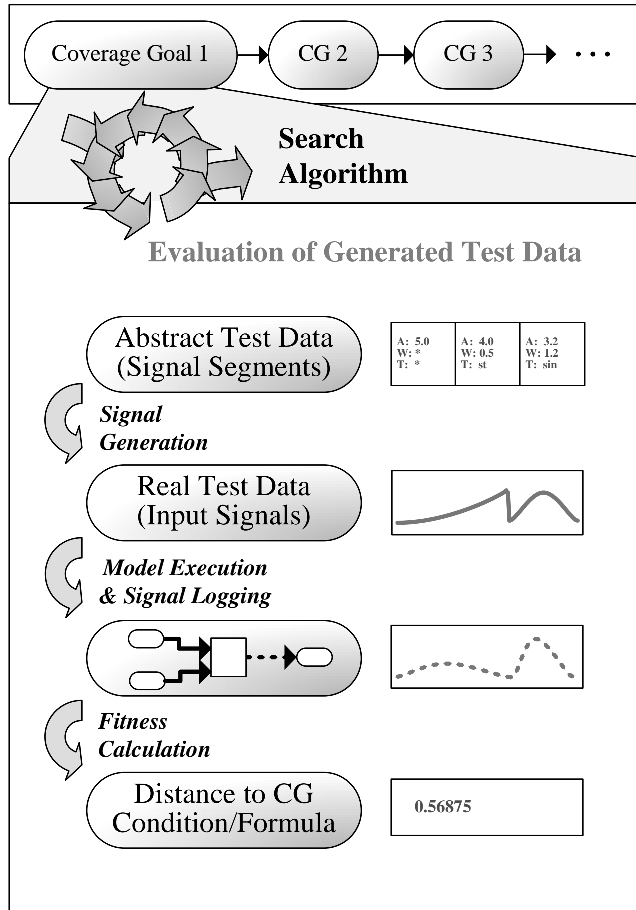


Figure 1. Automated search for test sequences, which fulfill coverage goals derived from the model under test.

the execution of a SL model involves the execution of every included block. Hence, a CG-related fitness function addresses only distances to the desired values of one or more model internal signals. For CGs in SF diagrams however, a bipartite fitness approach is possible [8]. Regardless of this, a fitness function has to operate on a sequence (signal) of distance values since distance calculations are done for every time step of the model's execution. Thus, the minimum value of a fitness signal is usually taken as final fitness value.

Figure 1 visualizes the overall work flow of applying search-based test data generation to structural testing of SL models as described.

C. Deficiencies and Potential

Search-based structural testing has been applied successfully to real (proprietary) SL models originating from development projects at Daimler, e.g., a model of a windscreen wiper controller [8]. Compared to purely randomized test data generation of similar complexity, the search-based approach results in significantly higher model coverage. Even in comparison with a commercial tool, the search-based approach performs more effectively.

Despite promising results, the approach lacks efficiency. In general, the overall runtime of the search processes for achieving maximal model coverage increases with the size of the model under test. Similar experiences have been made with code-level search-based structural testing [14]. Since a single automotive SL model is often hundreds of blocks in size, and because a test data generation process of more than a couple of hours is undesirable, improving efficiency of the search-based approach is vital. The following shortcomings of the search-based testing approach for SL, as proposed by Windisch [8], were identified as contributors to efficiency problems. They will be addressed by the work presented in this paper.

1) Even if a model is implementing its desired functionality entirely correctly, there are often model states that are simply unreachable. A search for test data that results in such model states cannot possibly succeed. However, the search technique is not aware of this and carries out a pointless and time-consuming search process.
2) Narrowing the size of the search space, i.e., the space of all possible input data, is crucial for how easy or difficult it is for a search to succeed. Industrial-sized SL models usually have many inputs for which suitable signals need to be found. The size of the overall search space varies with the number of model inputs. Reaching certain model states, however, is often independent of the stimulation of some of the model's inputs.
3) Targeting coverage criteria implies having to reach various CGs, between which, in fact, logical dependancies exist. The coverage of a CG often implies the coverage of other CGs (collateral coverage). In principle, each CG requires a separate search. Those search processes, however, are carried out in an uncontrolled order, regardless of how much collateral coverage they might cause.

There are two further technical problems leading to a lack of efficiency which are only partially addressed by the work presented in this paper. First, the structural test data generation is performed black-box-like, which means that the model is fed with input values on one end while some distances for calculating fitness are measured at some other point in the model. Any structural information between is not considered, thus the search might be blind to complicated dependencies in the model (cf. [15]). Second, when targeting a Boolean state in a model, a suitable fitness function is hard to find since a simple true or false rating inadequately leads a search [8]. Zhan and Clark suggest a technique called tracing and deducing [16], which mitigates this problem in certain cases, but fails in instances where the Boolean problem

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

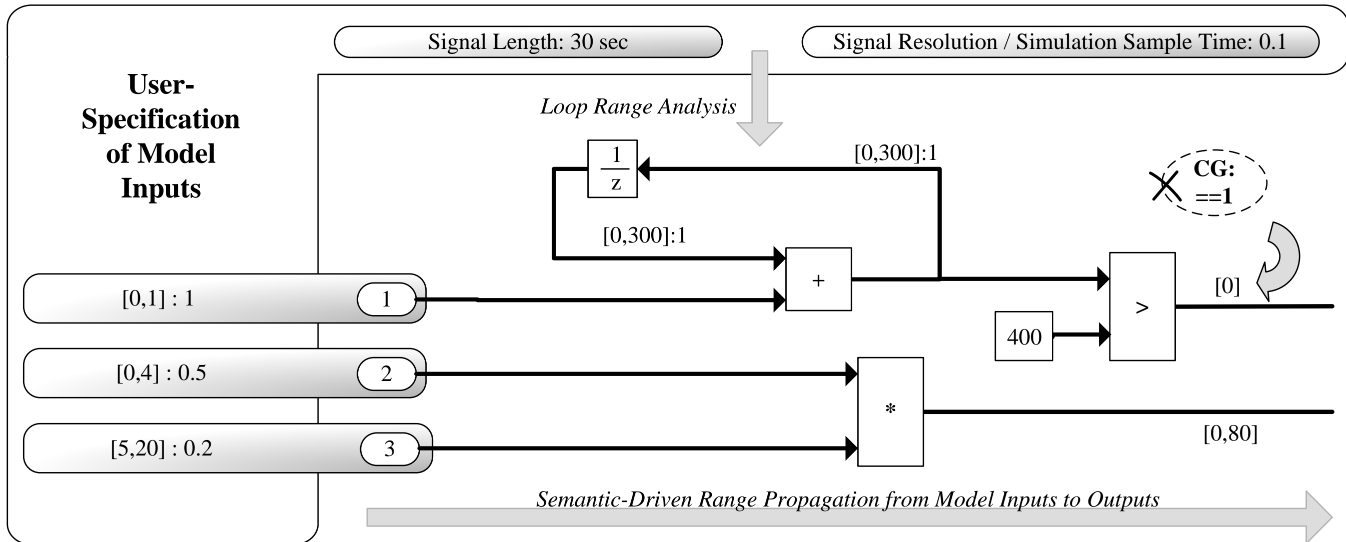


Figure 2. Example of how determining the ranges of a model's internal signals based on input specification and block semantics works.

cannot be traced back in the model to a non-Boolean one.

As a whole, we aim to improve the search-based approach for structural testing of SL models so that it performs acceptably and reliably in industrial development environments. To this end, we turn our attention to testing of TL-compliant SL models since the code generator TL is widely used in industrial practice. TL extends SL by offering additional block types, but also makes restrictions on the usage of certain SL constructs like block types. Nevertheless, it is possible to adapt our ideas to pure SL usage.

III. Static Preprocessing

We distinguish between techniques which support search-based structural testing (a) before the CG-related search processes, (b) between the different search processes, (c) during each search process, and (d) after the search processes are done. In the following sections, three techniques belonging to category (a) are presented. Apart from making use of an input specification and choice of coverage criteria provided by the user, all three techniques are fully automatic.

A. Signal Interval Analysis

In structural testing practice, achieving 100% coverage is often not possible. One reason lies in the semantic constructions precluding certain states or signal values. It might also be that a tester specifies the test data to be generated in such a way that it prevents certain CGs from being satisfiable. Also, SL models might be designed variably, e.g., contain a constant block with a variable value. When such variability is bound during execution, e.g., via configuration file, certain model states may be unreachable.

CGs referring to unreachable states worsen the overall runtime and undermine the efficiency of search-based structural test data generation since time-consuming search processes are carried out without any hope of finding desired test data. Therefore, we propose two techniques contributing to automatic identification of unreachable CGs. The first one is an interval analysis, which determines the range within which the values of every internal model signal are. If a signal range is in conflict with the range or value required by a CG, this CG is unsatisfiable. We use interval analysis since other approaches to detect infeasibility, such as constraint solving or theorem proving [17], are currently not scalable enough for the complex equations constituted by industrial-sized SL models. The second technique is an analysis of dependencies between CGs. Since this technique is mainly used for another purpose, it is presented in Section III-C.

The code generator TL, as well as the latest version of SL, are capable of analyzing signal ranges in order to perform code optimizations and improve scaling or data type selection, respectively. While those range analysis features are limited (e.g., determining ranges of signals that are involved in loops is not possible without user interaction) our signal interval analysis (SIA) makes use of an input signal specification in order to overcome such limitations and derive more precise ranges.

As mentioned in Section II-B, a tester who uses the search-based approach for testing SL models, as outlined by Windisch, is asked to specify the test data to be generated first. This involves establishing (a) the range boundaries and step size of each model input, as well as defining (b) a common length (in seconds) and sample rate for all input signals - $sig_{Length}, sig_{Res} \in \mathbb{R}^+$, with sig_{Length} being a multiple of sig_{Res}. SIA starts with information (a) for the model's input signals and propagates the corresponding signal ranges of the form $[x, y]{:}q$, where q (optional) is the step size, through the whole model. For every model

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

internal signal s_i, we keep a list of consecutive time intervals (time phases) of the form $[t_a, t_b]$, where $t_a, t_b \in \mathbb{N}_0$ and $0 \leq t_a \leq t_b \leq (sig_{Length}/sig_{Res})$. Every time interval goes along with an interval set $I(s_i)_{[t_a,t_b]}$ that contains the actual ranges. We use interval sets instead of a single interval per signal, or per time phase, in order to derive more accurate range information - as suggested by Wang et al. [18].

The propagation technique processes all model blocks in a predetermined order, which is equivalent to the block execution order that SL calculates for running a simulation. Each propagation step is based on the semantics of a block and the ranges of its incoming signals. The result of such a step are ranges (intervals) for the outgoing signals of the block. In this context, we derived interval semantics for each block type of TL-compliant SL models using basic concepts of interval arithmetic [19]. This approach can also be described as a form of abstract interpretation.

Example: The original semantics of a *Sum* block with two incoming signals s_1 and s_2 and the outgoing signal s_3 is $s_{3,t} = s_{1,t}+s_{2,t}$, where t is a time step. Given $I(s_1)_{[t_1,t_2]} = \{[a_{1,1}, b_{1,1}], ..., [a_{1,n}, b_{1,n}]\}$ and $I(s_2)_{[t_3,t_4]} = \{[a_{2,1}, b_{2,1}], ..., [a_{2,m}, b_{2,m}]\}$ with equal or overlapping time intervals, basic interval arithmetic is used to obtain the corresponding interval set for s_3: $I(s_3)_{[max(t_1,t_3),min(t_2,t_4)]} = \{[a_{1,1}+a_{2,1}, b_{1,1}+b_{2,1}]$, ..., $[a_{1,1}+a_{2,m}, b_{1,1}+b_{2,m}]$, ..., $[a_{1,n}+a_{2,1}, b_{1,n}+b_{2,1}]$, ..., $[a_{1,n}+a_{2,m}, b_{1,n}+b_{2,m}]\}$. One could, however, calculate only one resulting interval with the overall minimum and maximum boundary values of the intervals listed above in $I(s_3)$. Since this would lead to a loss of precision, we avoid this approach. In order to still keep interval sets small, we developed an algorithm that merges intervals of the very same set in a suitable way, where possible.

Figure 2 graphically depicts the overall procedure with the aid of a simple example. Note that the model contains a loop, initiated by a delay block with an initial value of 0. The standard propagation procedure would be unable to continue here since ranges are not available for all incoming signals of the sum block. A simple, yet imprecise solution is to set the range of the sum block's outgoing signal to the minimum and maximum values of the signal's data type. A more precise solution, however, is to use the information (b) of the signal specification in order to run a loop analysis. From length and sample rate, the number of loop iterations is derivable. Starting with the initial value of the delay block a static analysis of the loop iterations is performed, resulting in time-related range information. In order to keep the final results clean and minimal, as mentioned before, each signal's ranges as well as the time phases of ranges are combined, if possible, in each iteration of the loop analysis. Note that the ranges in Figure 2 are displayed simplified and summarized, omitting time intervals and the details of their interval sets.

Using range propagation and loop analysis in combination, SIA is capable of determining the ranges of all signals contained in the model under test. In cases of blocks with unknown semantics or unsupported blocks, the minimum and maximum values of the outgoing signal's data type are used. In the end, the results of SIA are used to assess whether each CG's associated formula is unsatisfiable - such as the CG in Figure 2. In addition to unsatisfiable CGs, SIA can also help in identifying Boolean signals or discrete signals with only a few possible different values. As described in Section II-C, CGs related to such signals can be problematic for the search-based approach.

B. Signal Dependency Analysis

By default, the search algorithm generates test data for all of the model's inputs when targeting a CG. However, there are usually CGs whose satisfaction is, in fact, independent of the stimulation of certain model inputs. By not taking this into account, the search space is unnecessarily large, which makes it more difficult for the search to find desired test data. To raise efficiency, we include a signal dependency analysis (SDA) to identify which model inputs each CG actually depends on. McMinn et al. [20] investigated a related approach, however, on code level. SDA is closely related to a slicing approach for SL models developed parallel to our work [21].

At code level, such analysis is usually done by capturing the control dependence in a graph. SL models though, as pointed out previously, are dominated by data dependencies. We therefore analyze the dependency of CGs on input signals by creating a signal dependency graph (a) based on the syntax of the model and (b) refined according to the semantics of blocks. Focusing purely on syntax, the following principle leads to a graph describing which signal b the value of a model internal signal a depends on: Signal a is dominated by a signal b if signal a is the outcome of a model block which has signal b incoming - written $a \rightarrow b$. Some blocks with multiple outgoing signals however, do not use every incoming signal in order to calculate the value of a certain outgoing signal. In such cases, the signal dependency graph is refined by removing over-approximated dependencies. Similar to SIA, SDA processes all model blocks in a predetermined order for collecting dependency relations.

In addition to the basic procedure of SDA as outlined above, some modeling constructs available in SL require special handling. The concept of vector signals, for example, which makes a signal being a container for a number of subordinate signals, requires tracing the dependency between subordinate signals of different containing signals. At this point, the semantics of blocks that process vector signals is relevant. A *Sum* block which has two vector signals as inputs, for example, performs a pair-wise addition of vector elements at the same vector index. Consequently, the resulting signal is also a vector. Each of its subordinate signals, however, is dominated by only two subordinate

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

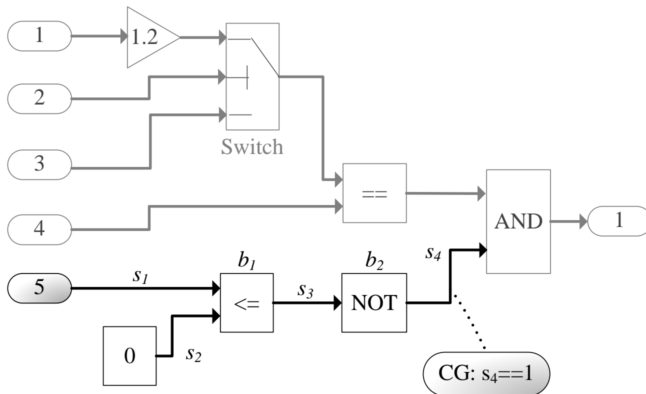


Figure 3. Illustration of how determining which model inputs a CG depends on might narrow the search space.

signals of the block's input signals. SDA's block-specific collecting of dependency relations considers such cases. A further exception that requires special handling by SDA is the concept of conditional subsystems in SL. Such subsystems are only executed if their control signals activate them or keep them active. Every signal inside of such a subsystem thus also depends on the control signal. SDA recognizes these situations when processing the *Inport* blocks within a conditional subsystem and adds dependency relations to the graph accordingly.

In order to finally determine which model inputs a certain CG depends on, the signal or signals which the CG expression refers to are selected in the dependency graph. By traversing the graph up to the input signals, the set of relevant model inputs is built up. Within the subsequent search process for this CG, signals are generated only for the relevant inputs. In addition, operators of the applied search algorithm which merge and modify generated solutions, such as crossover and mutation operators in a genetic algorithm, target only relevant input signals. In this way, depending on the specific application case, the search space might be reduced by several dimensions. For model execution, all other (irrelevant) model inputs receive a random signal that is consistent with the input's specification.

Figure 3 shows an example illustrating the beneficial potential of such an approach in the context of search-based test data generation. SDA's block analysis would build up a dependency graph expressing that $s_3 \rightarrow s_1$, $s_3 \rightarrow s_2$ (gathered by analyzing b_1), and $s_4 \rightarrow s_3$ (gathered by analyzing b_2). These relations alone are sufficient to discover that CG $s_4 == 1$ depends solely on input 5. Hence, varying signals for input 1 to 4 during a search has no effect on satisfying the targeted CG. If the search would focus on exploring the search space of input 5, however, it would likely find the desired input data sooner.

C. Coverage Goal Sequencing

No matter if structural testing is performed in addition to functional (black-box) testing or purely as white-box testing, it is usually a set of CGs that constitutes the test objective. Remember, that for each CG a separate search needs to be run. In Windisch's approach, those search processes are executed in random order. Hence, correlations between CGs are ignored. Given CGs with the expressions $s<90$, $s<80$ and $s<70$, for example, it is most likely more efficient to aim for reaching the goal $s<70$ first because it satisfies all other CGs at the same time. As this example indicates, the execution order of the CG-related search processes affects the efficiency of the whole structural test.

Other researchers in the search-based testing community have noticed this shortcoming as well. Fraser and Arcuri [22] advise focusing on the generation of whole test suites rather than targeting single CGs. They recommend optimizing multiple test suites instead of multiple test data and also suggest rewarding smaller test suites with a better fitness, in case two or more test suites achieve the same coverage. Harman et al. [23], in contrast, suggest a multi-objective search in which each CG is still targeted individually but the number of collateral (accidentally covered) goals is included as a secondary objective. Though facing a similar problem, our approach differs. We keep the focus on CGs themselves since, considering the complexity of the optimization problems constituted by industrial SL models, they are often difficult to reach and we do not want to impede the search by burdening it with additional goals or mixed fitness values. Instead, we propose a coverage goal sequencing (CGSeq) approach that creates a reasonable order in which the various CGs are pursued. Li et al. worked out a related approach [24], however, it is outside of the search-based and SL context. Ultimately, by maximizing collateral coverage, our approach attempts to minimize the number of CGs that need to be pursued. Not only is this expected to improve overall efficiency, but the resulting test suite should also be smaller.

The procedure of CGSeq is summarized in Figure 4. First of all, the model under test is analyzed and CGs are derived for all SL/SF-relevant coverage criteria (see overview by Windisch [8]). In preparation to analyzing dependencies between CGs, we apply several harmonization and simplification steps to the CG expressions. Note that results of SIA (Section III-A) are used for this task as well, e.g., an expression $s \geq 1$ would be transformed to $s=1$ if s is a Boolean signal.

Next, possible dependencies between CG expressions are analyzed, resulting in a dependency graph. In this graph, we treat the nodes as CG sets, in which equivalent CGs are grouped - noted as $CGN = \{CG_a, ..., CG_n\}$. Note that we omit the set braces in some of the following notations if the set contains only one element. In order to limit graph complexity, the dependency analysis considers only

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

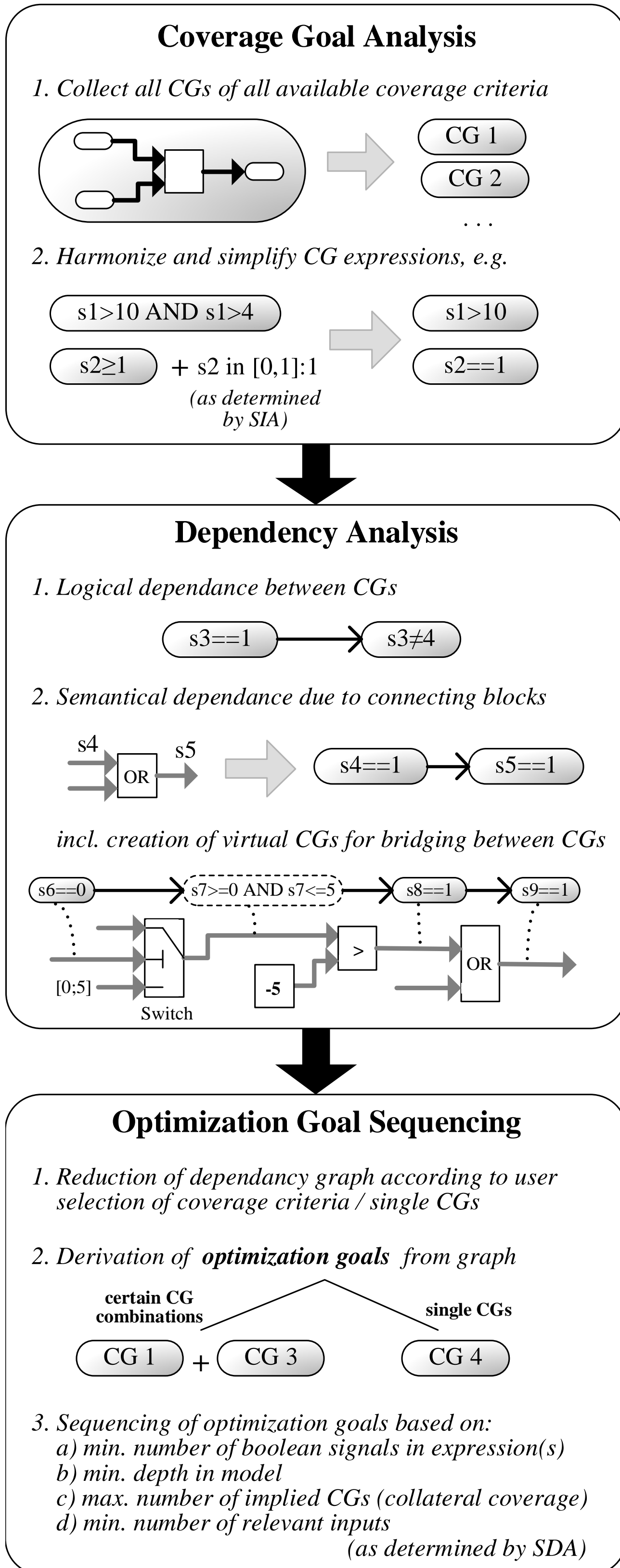


Figure 4. Main process steps to create an efficient order for a set of coverage goals which are processed separately by a search.

relations that are useful for assessing CG satisfiability and for the final goal of sequencing the CGs. The following relations are captured: implication ($CGN_1 \rightarrow CGN_2$), equivalence ($CGN_1 \cup CGN_2$), NAND ($CGN_1 \uparrow CGN_2$), and XOR ($CGN_1 \oplus CGN_2$). For a more compact notation of implications we also use, in certain cases, conjunctions $((CGN_1 \wedge ... \wedge CGN_n) \rightarrow (CGN_m \wedge ... \wedge CGN_z))$ and disjunctions $((CGN_1 \vee ... \vee CGN_n) \rightarrow (CGN_m \wedge ... \wedge CGN_z))$. Dependencies between CG expressions are analyzed both from a logical and a semantical point of view.

Logical dependency addresses relations between CGs due to the operators involved in their formulas, as well as due to contained constants and the ranges of related signals, if SIA has been carried out prior to CGSeq. For example, the relation $(s_1{>}0)_{CG} \rightarrow (s_1{\geq}0)_{CG}$ is recognized solely based on the involved operators. The relation $(s_1{==}5)_{CG} \rightarrow (s_1{\geq}0)_{CG}$, however, requires taking the involved constants into account. Going one step further, the relation $(s_1{==}3)_{CG} \rightarrow (s_1{>}s_2)_{CG}$ would be detected if $max(I(s_2)){<}3$, i.e., the maximum upper boundary of any interval in s_2's range set is less than the constant 3. An extensive distinction of such cases has been worked out.

Semantical dependency means that for each two CGs relating to incoming or outgoing signals of the same model block, a block-specific analysis checks if a relation between the CGs exists. Just as within SIA and SDA, all model blocks are processed in a predetermined order for this analysis. Given an *OR* block with the incoming signals s_1 and s_2, and the outgoing signal s_3, the analysis for this block would detect, for example, the relations $(s_1{\neq}0)_{CG} \rightarrow (s_3{==}1)_{CG}$, $(s_2{\neq}0)_{CG} \rightarrow (s_3{==}1)_{CG}$, and $((s_1{==}0)_{CG} \wedge (s_2{==}0)_{CG}) \rightarrow (s_3{==}0)_{CG}$. In certain cases the block-specific analysis adds virtual CGs as a bridge to other CGs in order to detect further dependencies. For this purpose, as illustrated in Figure 4, signal range information from SIA is also used in case of certain block types.

As a side effect, based on the captured dependencies, further CGs might be detected as unsatisfiable in the course of CGSeq (cf. Section III-A). For example, if the graph contains the relation $CG_1 \leftrightarrow CG_2$ and SIA has discovered that CG_2 is unsatisfiable, then CG_1 must also be unsatisfiable. CGSeq performs an extended satisfiability check by propagating unsatisfiability conclusions through the graph.

Now, back to the goal of sequencing the CGs. In the next step, the user's selection of coverage criteria or single CGs is considered by minimizing the graph accordingly. Amongst others, non-selected CGs implying selected CGs are kept. Afterwards, the final optimization goals are derived from the graph in a two-fold way:

1) For each selected CG, called $CG_{Original}$, an optimization goal consisting of a single CG, called CG_{Target}, is derived. Starting in the dependency graph at the node of $CG_{Original}$, the implication relations are analyzed backwards to determine CG_{Target}, which

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

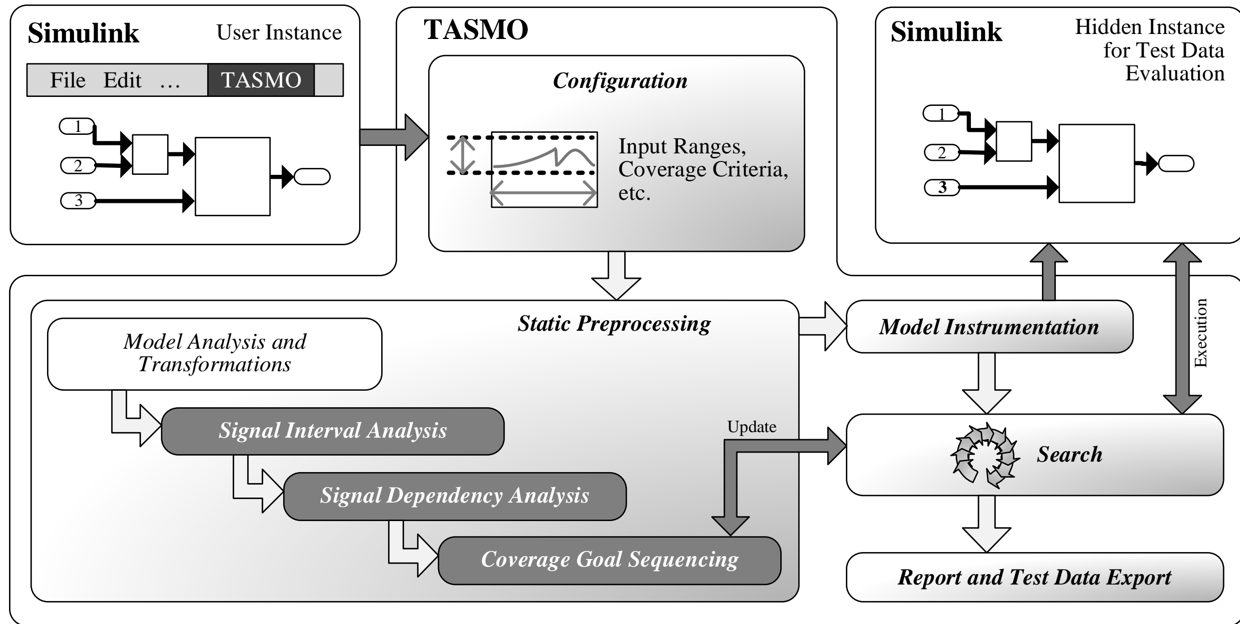


Figure 5. Overall workflow of the tool prototype for search-based test data generation for Simulink, including the static preprocessing workflow.

ultimately is representing $CG_{Original}$. If multiple CGs are suitable candidates for CG_{Target}, criteria such as a lower model depth (minimum path length from any model input to any signal involved in the CG) or a lower rate of Boolean signals involved in the CG expression are taken into account.

2) Certain combinations of CGs are derived to form optimization goals as well. Remember that the dependency graph can contain conjunctions. Each conjunction serves as a start point for building up a conjunction tree which contains implication relations from the graph that are leading (directly or indirectly) to the conjunction. Out of every branch or sub-branch within this tree one suitable CG is selected. These CGs constitute one or multiple new optimization goals, depending on the depth of the tree. In this way, a few suitable optimization goals with potentially high collateral coverage are added.

Finally, the optimization goals are sequenced according to several metrics, primarily by the number of (so far unsatisfied) implied CGs, but also by their depth in the model and the amount of Boolean signals involved in the expressions - since such goals should be avoided given the fitness function construction problem (see Section II-C). Note that the pursuing order of optimization goals is updated after each search process ends, since an optimization goal's number of unsatisfied implied CGs might have changed.

IV. Implementation

The presented preprocessing techniques have been implemented in the course of developing our prototypical tool *TASMO* (Testing via Automated Search for Models) [25].

TASMO is mainly written in Java and closely integrated with Matlab. As shown in Figure 5, the user can trigger the automated test data generation process directly from within Matlab when having a SL model opened. *TASMO* then extracts model related information using Matlab's API and programming language *m*. After the Java-based component of *TASMO* has been triggered, it builds up an internal representation of the model under test and derives all CGs for every supported coverage criteria from this representation in advance. It then applies transformation and reduction steps to the internal model representation in order to focus on the relevant parts for structural test data generation. In particular, all blocks and signals from the model's *Outports* to the rightmost CGs are removed since they are irrelevant for the following analysis and preprocessing steps. Model constructs like virtual subsystems or bus systems, which are both semantically irrelevant, are flattened.

The user is then asked to select one or more coverage criteria, like decision coverage or condition coverage, or certain CGs directly. Regarding this step, an insight into the user interface of the tool is given in Figure 6. As indicated earlier, the user is also asked to specify the test data to be generated, e.g., the range of each model input. After all required presettings have been made, the three presented preprocessing techniques are run in the following order: First, SIA determines the ranges of all model internal signals in order to assess the satisfiability of each CG. During SIA, *TASMO* might also transform the internal model representation. Both general and block-specific transformation

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

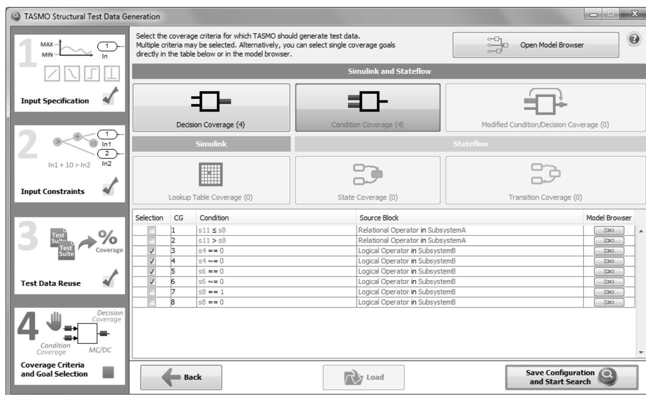


Figure 6. Automated testing of Simulink models with *TASMO*: Selection of coverage criteria and coverage goals for the model under test.

criteria are checked. For example, if a signal's ranges all contain only a single constant value and this signal does not originate from a *Constant* block, the source block of this signal is replaced with a *Constant* block. At the same time, a backwards directed removal process is initiated starting with the block that has been replaced. All signals and blocks preceding this block (directly or indirectly) are removed if no CGs are referenced to it. In this way, the internal model is kept as compact as possible for subsequent analysis steps without altering its semantics. Another example: If the incoming signal of an *Abs* block (absolute value) has solely non-negative ranges, the block can be removed and its former incoming signal can be directly connected to its former outgoing signal.

After SIA, *TASMO* runs SDA in order to determine for each CG which model inputs it depends on. Finally, CGSeq is carried out to build up and sequence a list of optimization goals. Before the search algorithm processes optimization goal after optimization goal, a hidden Matlab instance for simulating the model with generated test data is started. A copy of the real model under test is loaded, initialized, and instrumented. Afterwards, an automated search for each optimization goal is performed, which generates test data solely compliant with the signal specification for all inputs that are relevant for the CGs of the current optimization goal. If an optimization goal, or rather its contained CGs, are covered by accident during a search for another optimization goal (collateral coverage), the goal is considered done. When finished, *TASMO* generates a report and provides the generated test data in a reusable format.

V. Case Study

We investigated the effect of the three preprocessing techniques SIA, SDA and CGSeq on structural test data generation for industrial SL/TL models. Two SL/TL models served as case examples, model A and B.

Model A originates from the development of an electric vehicle's propulsion strategy, contains 730 blocks, and has 12 inputs. A total of 600 CGs were derived for the chosen coverage criteria *decision coverage* and *condition coverage*. Model B implements the functionality of a rear window defroster. It contains 1861 blocks, has 16 inputs, and 1113 CGs were derived in total.

Table I
Case Study Configurations

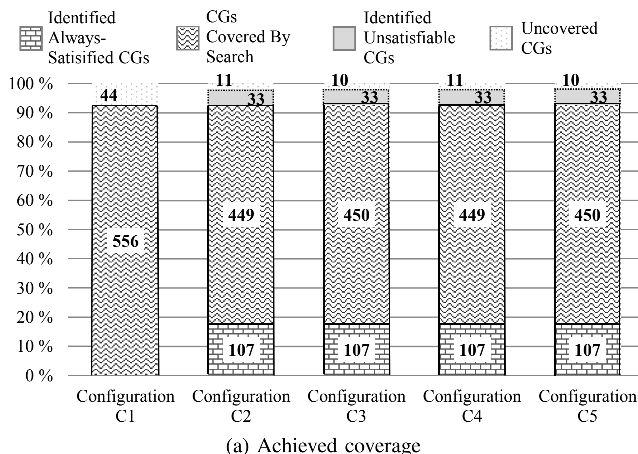
Configuration	SIA	SDA	CGSeq
C1	-	-	-
C2	x	-	-
C3	x	x	-
C4	x	-	x
C5	x	x	x

In order to analyze the effect of the presented techniques, five different configurations were compared. Table I outlines the characteristics of each configuration. The configurations differ in the use or non-use of the static preprocessing techniques within the test data generation process. C2 is compared with C1 to evaluate the effect of SIA, C3 is compared with C2 to evaluate the effect of SDA, and C4 is compared with C2 to evaluate the effect of CGSeq. C5 brings all three static preprocessing techniques together to evaluate the entire static preprocessing as a whole. In particular, only C5 allows CGSeq to use the results of both SIA and SDA in order to analyze dependencies between the coverage goals and to derive an execution order for the search goals. Note that SIA is also activated in C3 and C4, since processing unsatisfiable CGs would otherwise extend the runtime of the case study unnecessarily.

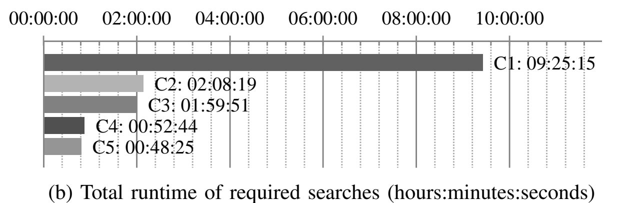
For the search, we applied a special genetic algorithm for generating signals, as presented by Windisch and Al Moubayed [13] (see Section II-B). The algorithm settings were chosen as listed in detail in their paper, except for the following differences. In each search iteration, 20 individuals (test data) were generated. A search was stopped when the targeted optimization goal was reached, when 40 iterations had been carried out, or if the search stagnated. Search stagnation was indicated by 8 successive iterations in which no better individual was found. Since the search algorithm is subjected to a certain randomness, we carried out a total of 30 test data generation runs for each configuration (C1-C5). The input signals to be generated for model A were specified to have a length of 30 seconds and a sample rate of 0.1 seconds. For model B, the signal length was set to 20 seconds and the sample rate to 0.25 seconds. While the signal lengths were chosen with respect to the dynamic functional behavior of the models, the sample rates are specified in each model's properties. Range boundaries for each model's inputs (see Section II-B and III-A) were taken directly from development documents.

The study was run on a PC with an Intel Core 2 Duo processor (P8600, 2.4 GHz), 4 GB RAM, and Windows 7

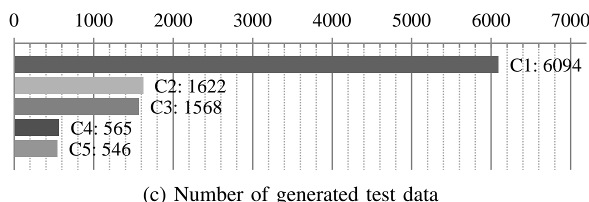
2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



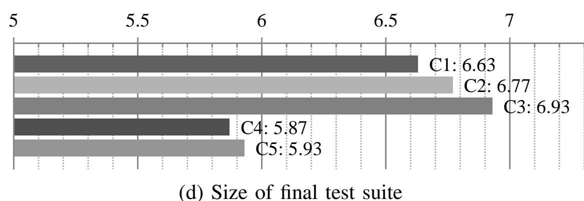
(a) Achieved coverage



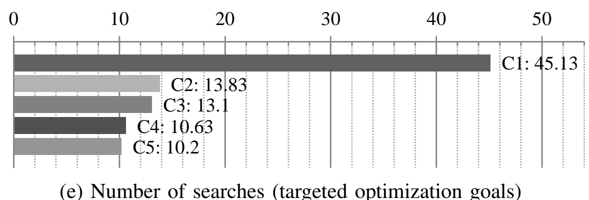
(b) Total runtime of required searches (hours:minutes:seconds)



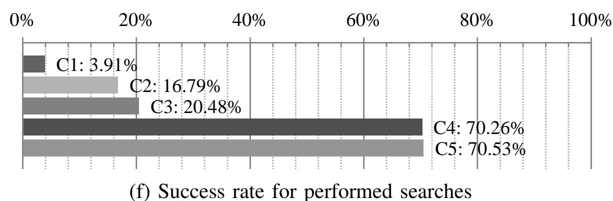
(c) Number of generated test data



(d) Size of final test suite

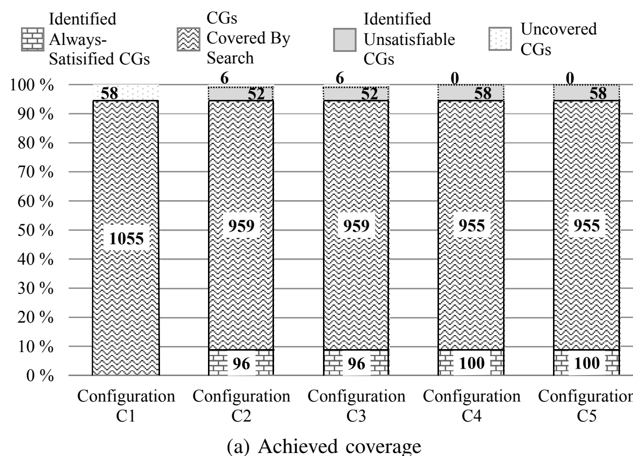


(e) Number of searches (targeted optimization goals)

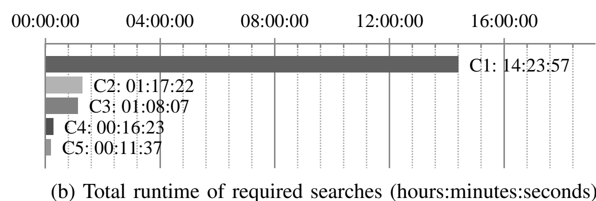


(f) Success rate for performed searches

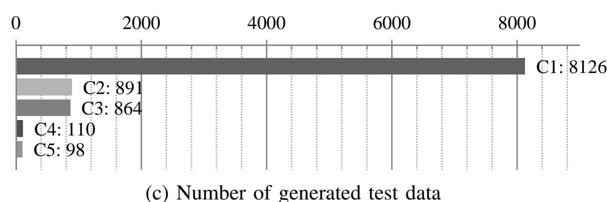
Figure 7. Measured variables of test data generation for model A, averaged over 30 test data generation runs.



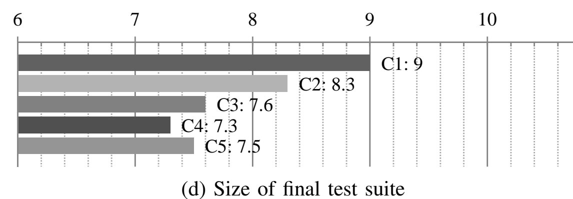
(a) Achieved coverage



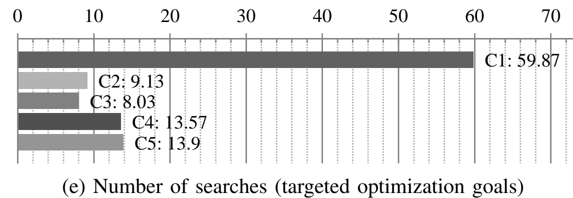
(b) Total runtime of required searches (hours:minutes:seconds)



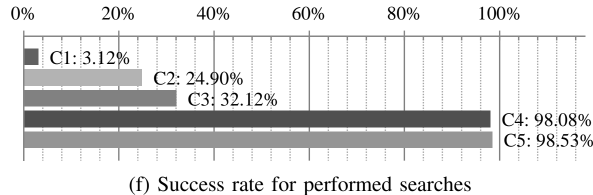
(c) Number of generated test data



(d) Size of final test suite



(e) Number of searches (targeted optimization goals)



(f) Success rate for performed searches

Figure 8. Measured variables of test data generation for model B, averaged over 30 test data generation runs.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

64-bit. Our Java-based tool *TASMO* was run using version 7 of the Java Runtime Environment in connection with Matlab/Simulink 2009b and TargetLink 3.1.

In the following sections, the results of the case study are presented. First, we analyze the results with regard to each preprocessing technique separately. Then, we assess the use of our preprocessing techniques as a whole.

A. Analysis of SIA

As visible in Figure 7a for C2, SIA identified 33 unsatisfiable CGs for model A, of which 8 were identified by SIA's loop analysis. In addition, 69 CGs from a total of 107 CGs, which are always satisfied independent of the chosen input data, were identified by SIA. All those CGs were excluded from subsequent search processes. As for model B, 58 CGs are unsatisfiable, of which 52 were identified by SIA (6 by its loop analysis), as can be seen for C2 in Figure 8a. For model B, SIA also identified 96 CGs from a total of 100 CGs that are always satisfied. Note that the other always-satisfied or unsatisfied CGs were identified by CGSeq.

In order to evaluate the effect of preprocessing the test data search with SIA, C1 and C2 are analyzed. The model coverage achieved by a configuration is used to measure for effectiveness of a configuration. C1 was able to generate test data for model A with a coverage of almost 93% (95% for model B). While introducing SIA in C2 did not lead to higher coverage, as can be seen in Figures 7a and 8a, it certainly improved the efficiency of the automated test data generation process. As the statistics for model A in Figure 7b show, the runtime of all performed searches together was reduced by 77% (from 9:25 hours for C1 to 2:08 hours for C2). For model B, the runtime was even reduced by 91% (from 14:24 hours for C1 to 1:17 hours for C2, see Figure 8b). The runtime of SIA itself, which is not included in the values displayed in Figures 7b and 8b, was about 5 minutes for model A and 15 seconds for model B. In large part, SIA's runtime was caused by its loop analysis feature.

In C2, test data was only generated for CGs that were not identified as unsatisfiable. While the test data generation process of C1 for model A was targeting about 45 CGs directly by search (about 60 for model B), only 14 (9 for model B) were targeted by C2 (see Figures 7e and 8e). The number of searches that were carried out additionally by C1 is approximately as high as the number of unsatisfiable CGs that SIA was able to identify. As a consequence of C2 targeting less CGs, less test data was generated overall, as displayed in Figures 7c and 8c. Since C1 performed searches for unsatisfiable CGs without any hope for success, the portion of successfully tackled CGs is naturally higher for C2 in comparison to C1 (see Figures 7f and 8f).

B. Analysis of SDA

Using SDA, the search processes of C3 and C5 were ignoring irrelevant inputs during input data optimization. According to the preprocessing analysis carried out by SDA, on average, 4.18 of model A's 12 inputs turned out to be relevant for reaching one of the CGs (6.58 of 16 inputs for model B). Considering only the CGs that were targeted in C3, 6.48 of the 12 inputs of model A, on average, are relevant (12.06 of 16 for model B). For C5, the average value is 4.71 (model A) and 7.85 (model B), respectively. These values reflect the effective search space reduction due to the use of SDA.

Comparing the coverage results of C3 with those of C2, as well as C5 with C4, the introduction of SDA to the automated test data generation process for model A led to slightly higher coverage (see Figure 7a). One coverage goal that was only covered seldomly by the other configurations was always covered with SDA being activated. This observation is backed up by the increased rate of successfully finished searches for C3, as visible in Figure 7f. In case of model B, C1 managed to cover as many CGs as C2.

The use of SDA reduced the overall runtime of the searches by 7% for model A (from 2:08 hours for C2 to 1:59 hours for C3) and 12% for model B (from 1:17 hours for C2 to 1:08 hours for C3), as visible in Figures 7b and 8b. For both models, the runtime of SDA itself was only about one second. Due to focusing solely on relevant model inputs, the searches for a few targeted CGs finished slightly quicker. Accordingly, less test data was generated, as can be seen when comparing C3 with C2 in Figures 7c and 8c.

C. Analysis of CGSeq

Note that during CGSeq, another 38 always-satisfied CGs were identified for model A in addition to 69 such CGs that were already identified by SIA (107 in total as shown in Figure 7a). Likewise, 4 additional always-satisfied CGs were identified by CGSeq for model B, resulting overall in 100 such CGs (see Figure 8a). All these CGs were considered as covered and were excluded from the test data generation process since any test data in the generated test suite would satisfy them. For model B, CGSeq even found another 6 unsatisfiable CGs, in addition to 52 unsatisfiable CGs identified by SIA (see Figure 8a).

Using CGSeq prior to the test data search (as done in C4) did not increase the total model coverage (compared to C2), as apparent in Figures 7a and 8a. However, the runtime of the search was reduced by 59% for model A (from 2:08 hours for C2 to 52 minutes for C4) and 79% for model B (from 1:17 hours for C2 to 16 minutes for C4) due to introducing CGSeq in the test data generation process, as visible in Figures 7b and 8b. Similar to SIA and SDA, CGSeq itself did turn out to be reasonable in terms of computation time. Running CGSeq took only about 16 seconds for model A and 20 seconds for model B on average.

Due to targeting more promising (e.g., non-boolean) goals first, the portion of successfully tackled searches was higher (see Figures 7f and 8f) and less test data had to be generated

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

(see Figures 7c and 8c). For model A, less searches were performed (compare C4 and C2 in Figure 7e) and the automated test data generation process resulted in smaller test suites (see Figure 7d) since CGSeq prioritizes goals with potentially high collateral coverage. While the size of the resulting test suite was also reduced when activating CGSeq for model B (compare C4 and C2 in Figure 8d), more searches were performed (see Figure 8e). This might be surprising at first sight. However, it demonstrates how the test data generation process becomes more target-oriented when using CGSeq. In C2, a lot of CGs that are difficult to approach for the search algorithm were targeted. Plenty of test data was generated in the course of this and as a side effect, many CGs were covered rather coincidentally.

D. Combined Analysis

In summary, all three preprocessing techniques have demonstrated their usefulness to automated test data generation for SL/TL models. The case studies have shown that SIA is able to raise the test data generation's efficiency significantly, due to exclusion of unsatisfiable CGs. SDA pointed out its capability to raise both effectiveness and efficiency by turning the focus of the search for input data on relevant model inputs. CGSeq improved the efficiency considerably by advising the test data search to preferably target CGs for which a suitable fitness function is derivable, and which entail high collateral coverage. In addition, the runtime of all three preprocessing techniques was vanishingly low compared to the runtime of a subsequent test data search. Thus, we conclude that their use comes without any significant negative side effects.

Running all three prepocessing techniques in combination, as done in C5, indicates that the advantages of each technique also complement each other suitably. While in C3, only SIA and SDA, and in C4, only SIA and CGSeq were combined, the combination of all of them led to a further reduction of the search runtime of 8% for model A and 29% for model B (see Figures 7b and 8b). Even though the improvement in terms of runtime, as well as number of generated test data, performed searches, and ratio of successful searches, is only small (see Figure 7), C5 came out on top of all configurations for both effectiveness and efficiency of the automated test data generation, in case of model A as well as in the case of model B.

The case study also demonstrates the practicability and feasibility of search-based test data generation for SL/TL models, if extended by static preprocessing such as the presented ones. In particular, even if no automated test oracle is available to evaluate the generated test suite's conformance to the specification, and the test suite thus needs to be analyzed manually under a functional aspect, the case study has shown that the applied test data generation approach can lead to relatively small test suites, making a manual analysis feasible. Note that applying test suite minimization techniques might further reduce the size of the obtained test suites.

Further improvements and extensions of the applied test data generation technique might increase its performance to industrial SL/TL models even more. We investigated, for instance, why certain CGs of model A were not covered at all during the case study. It turned out that 8 CGs (of 10 uncovered ones, see Figure 7a) are unsatisfiable and neither SIA nor CGSeq were able to identify this circumstance; mainly due to block types in the chosen model for which the preprocessing techniques did not offer a specific handling. The two satisfiable, yet uncovered CGs left, were found to be difficult to solve by the search-based algorithm. Further guidance, for instance, by advanced fitness functions, is required to facilitate the search to reach such CGs. Nevertheless, the work presented in this paper turned out to be a big step forward towards industrial applicability of search-based automation of structural testing for SL/TL models.

VI. Related Work

Besides search-based testing, as introduced in Section II, a few other approaches have been applied or proposed for automating structural test data generation for SL models. In general, the approaches can be classified as dynamic, static or hybrid techniques. Dynamic techniques, such as search-based test data generation, execute the test object during test data generation. Static techniques however, analyze the test object without executing it in order to generate test data. Hybrid techniques usually contain characteristics of both dynamic and static techniques. Our approach mainly utilizes a dynamic technique, but could also be classfied as a hybrid approach since it is combined with static techniques.

In the field of structural test data generation for SL, the following dynamic techniques have been applied: search-based testing and random testing. Case studies of Windisch have shown that search-based testing outperforms random testing for structural testing of SL models [8]. As for static techniques, the use of symbolic execution/constraint solving and model checking has been reported in the SL context. Gadkari et al. developed an approach called *AutoMOTGen* [26], which involves the transformation of SL/SF models into a representation of the high-level language SAL and the use of a model checker for generating test data [27]. While the authors encountered promising results, they were also faced with technical limitations of their approach, particularly scalability issues. Păsăreanu et al. use symbolic execution and constraint solving in order to generate test data for SL models [28]. They also transform the model first - in their case, into Java code. The tool *Symbolic PathFinder* is then used to generate test data. However, their reports indicate that this approach suffers from scalability issues. Satpathy et al. developed *REDIRECT*, which applies concolic testing to the test data generation problem for SL [29]. Similar to the work of Gadkari et al., they transform

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

SL/SF models to SAL first. Concolic testing extends test data generation via symbolic execution and constraint solving by random testing. *REDIRECT* could also be classified as a hybrid approach. The authors document satisfying results, however, the SL models used in their case studies focus on SF diagrams and are much smaller than the ones used in this work. Peranandam et al. went along a similar path and combined random testing, constraint solving, model checking, and heuristics in their test data generation tool *SmartTestGen* for SL/SF models [30]. *SmartTestGen* uses a classification algorithm that statically estimates which of the listed techniques is likely to cover the coverage goal in question and therefore applies it for test data generation. The authors conclude that a case-specific use of different test data generation techniques is advantageous to mastering the complexity of industrial-sized SL models and the heterogeneity of their test data generation problems.

Commercial tools for structural test data generation for SL models are available as well. *Reactis* [31] from Reactive Systems, *T-VEC* [32] from T-VEC Technologies, and *Simulink Design Verifier* [33] from The MathWorks are common tools for this job. Only little is published about how these tools generate test data. Windisch [8] asserted that all these tools use randomized test data generation in some way. While *Reactis* combines this basic technique with guided simulations, *T-VEC* with symbolic execution and constraint solving, and *Simulink Design Verifier* with static analysis, the details of their implemented techniques remain unknown.

While dynamic approaches might face efficiency issues at times, as described for search-based testing in Section II-C, or have difficulties covering certain structural goals, as in the case of randomized test data generation, purely static approaches seem to lack scalability. As demonstrated in this paper, we believe in the potential of hybridization, i.e., the extension of dynamic test data generation techniques with static techniques. Since search-based test data generation is basically advanced randomized test data generation, which is used as a basic technique in various approaches and tools, we continue following the search-based testing path. Note that our prototypical tool *TASMO* is also applying different test data generation strategies, such as constraint solving, when a CG is suitable. This option was deactivated during the case studies presented in this paper though. More generally, the presented static preprocessing techniques are even usable independent of the choice of test data generation technique. Identifying unsatisfiable CGs, focusing on relevant model inputs, and generating an order of satisfiable CGs for efficient processing, is also advantageous when the test data is generated by randomization, constraint solving, or hybrid techniques.

VII. Conclusion and Future Work

This paper introduces an approach to improving the performance of search-based testing when applied to structural testing of SL models. Three static techniques extend the standard search-based approach by analyzing the model under test before the search processes for each CG are run. Unsatisfiable CGs are partially identified and excluded from the search. The search space is reduced in such a way that the search focuses solely on relevant model inputs. The separate search processes for each CG are sequenced in order to maximize collateral coverage, minimize test suite size, and shorten the overall search runtime.

A tool prototype that demonstrates the applicability of the extended search-based approach in industry has been developed. A case study with two industrial SL/TL models from the automotive domain has been performed, demonstrating how the presented preprocessing techniques improve the search-based test data generation approach. In particular, efficiency was raised distinctly. The use of all three techniques in combination led to a reduction of over 90% in the runtime otherwise required for search-based test data generation without the use of any of these techniques.

Despite satisfactory results, further adaption of the tool to industrial requirements is required, e.g., improved support of SL/TL blocks and SF diagrams by the presented static preprocessing techniques. Our next main step is to work on an expanded hybridization of the test data generation process. Besides integrating constraint solving techniques, we plan on using different types of search algorithms in combination, while factoring more information collected during model execution into the (hybridized) search algorithm. Finally, a broader comparison of the approach and tool, in particular with established commercial tools, is required.

References

[1] B. Wilmes, “Automated structural testing of Simulink / TargetLink models via search-based testing assisted by prior-search static analysis,” in *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, 2012, pp. 51–56.

[2] The Mathworks, “Matlab Simulink,” Last access: 2013-08-16. [Online]. Available: http://www.mathworks.com

[3] dSpace, “Targetlink,” Last access: 2013-08-16. [Online]. Available: http://www.dspace.com

[4] P. McMinn, “Search-based software testing: Past, present and future,” in *IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW ’11, 2011, pp. 153–163.

[5] B. Wilmes, A. Windisch, and F. Lindlar, “Suchbasierter Test für den industriellen Einsatz,” in *4. Symposium Testen im System- und Software Life-Cycle*, 2011.

[6] Y. Zhan and J. A. Clark, “A search-based framework for automatic testing of MATLAB/Simulink models,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 262–285, Feb. 2008.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[7] A. Windisch, "Search-based testing of complex Simulink models containing Stateflow diagrams," in *31st International Conference on Software Engineering*, 2009, pp. 395–398.

[8] A. Windisch, "Suchbasierter Strukturtest für Simulink Modelle," Ph.D. dissertation, Berlin Institute of Technology, 2011.

[9] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, May 1976.

[10] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[11] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, pp. 226–247, 2010.

[12] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[13] A. Windisch and N. Al Moubayed, "Signal generation for search-based testing of continuous systems," in *International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '09, 2009, pp. 121–130.

[14] T. E. Vos, A. I. Baars, F. F. Lindlar, P. M. Kruse, A. Windisch, and J. Wegener, "Industrial scaled automated structural testing with the evolutionary testing tool," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 175–184.

[15] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search based test data generation," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 13–24.

[16] Y. Zhan and J. A. Clark, "The state problem for test generation in Simulink," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2006, pp. 1941–1948.

[17] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: ACM, 1994, pp. 80–94.

[18] Y. Wang, Y. Gong, J. Chen, Q. Xiao, and Z. Yang, "An application of interval analysis in software static analysis," in *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, ser. EUC '08, vol. 2. Washington, DC, USA: IEEE Computer Society, 2008, pp. 367–372.

[19] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[20] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," *IEEE Transactions on Software Engineering*, vol. 38, pp. 453–477, 2012.

[21] R. Reicherdt and S. Glesner, "Slicing Matlab Simulink models," in *34th International Conference on Software Engineering*, 2012, pp. 551–561.

[22] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *11th International Conference on Quality Software*, 2011, pp. 31–40.

[23] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *3rd International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10, 2010, pp. 182–191.

[24] J. J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation," *Information and Software Technology*, vol. 48, no. 12, pp. 1187–1198, 2006.

[25] B. Wilmes, "Toward a tool for search-based testing of Simulink/TargetLink models," in *4th Symposium on Search Based Software Engineering (Fast Abstracts)*. Fondazione Bruno Kessler, 2012, pp. 49–54.

[26] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar, "AutoMOTGen: Automatic model oriented test generator for embedded control systems," in *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008, pp. 204–208.

[27] A. A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh, "Automatic generation of test-cases using model checking for SL/SF models," in *Proceedings of the 4th Model-Driven Engineering, Verification and Validation Workshop*, 2007, pp. 33–46.

[28] C. S. Păsăreanu *et al.*, "Model based analysis and test generation for flight software," in *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology*, 2009, pp. 83–90.

[29] M. Satpathy, A. Yeolekar, and S. Ramesh, "Randomized directed testing (REDIRECT) for Simulink/Stateflow models," in *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008, pp. 217–226.

[30] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh, "An integrated test generation tool for enhanced coverage of Simulink/Stateflow models," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 308–311.

[31] Reactive Systems, "Reactis," Last access: 2013-08-16. [Online]. Available: http://www.reactive-systems.com

[32] T-VEC Technologies, "T-VEC Tester for Simulink and Stateflow," Last access: 2013-08-16. [Online]. Available: http://www.t-vec.com/solutions/simulink.php

[33] The Mathworks, "Simulink Design Verifier," Last access: 2013-08-16. [Online]. Available: http://www.mathworks.de/products/sldesignverifier

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org