The State Problem for Test Generation in *Simulink*

Yuan Zhan
Department of Computer Science
University of York
York, YO10 5DD, UK
+44-1904-432749
yuan@cs.york.ac.uk

John A. Clark
Department of Computer Science
University of York
York, YO10 5DD, UK
+44-1904-433379
jac@cs.york.ac.uk

ABSTRACT
Search based test-data generation has proved successful for code-level testing. In this paper we investigate the application of such approaches at the higher levels of abstraction offered by *Matlab-Simulink* models. The presence of persistent state has been shown to be problematic at the code level and such difficulties remain when *Matlab-Simulink* models are to be tested. In such cases, sequences of inputs that can put the model under test into particular states are needed to enable the underlying test goals to be achieved. Simple search guidance appears to be insufficient and results in a 'flat' cost function landscape. To address this problem, we introduce a technique called ***tracing and deducing***, which helps provide better guidance to the search, allowing our developed tools to home in on the targeted test-data.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools (data generator), tracing.*

General Terms: Design, Verification.

Keywords: Matlab-Simulink, test-data generation, automation, structural coverage, state problem, tracing and deducing.

1. INTRODUCTION
The modern aim of 'testing' is to discover faults at the earliest possible stage as the cost of fixing an error increases with the time between its introduction and detection. Thus high-level models have become the focus of much modern-day verification effort and research. *Matlab/Simulink* [21] is a widely used notation in the dynamic system development industry that allows models to be created and exercised. *Matlab/Simulink* models are sometimes considered by industry as architectural level designs of software systems. The simulation facilities allow such models to be executed and observed. This property of *Simulink* turns out to be an advantage for effective dynamic testing. *Simulink* can serve many purposes in testing: as a model from which test data can be generated, as reference model for test coverage, as a source for the generation of test oracles, and as a test object in its own right. Many designers choose to model using *Simulink* and generate code automatically from its designs. Although it is possible to encode *Simulink* models in hardware, the production of high-level retargetable code (e.g. in Ada or C) is most popular.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

Other authors have recognized the practical significance of such modeling and the need to provide assurance information automatically, e.g. the worst-case execution times for such models [17]. Baresel et al. [9] proposed an innovative way of generating sequences of signals for testing *Simulink* models by building the overall signal from a series of simple signal types such as step, ramp and sine curves etc. We have focused on the dynamic test-data generation for *Simulink* models, for both structural coverage and mutation coverage criteria [13][22].

The presence of persistent state causes difficulty in automatic test-data generation not just at the code level [24], but also for *Matlab-Simulink* models. In such cases, sequences of inputs that can put the model under test into particular states are needed to enable the underlying test goals to be achieved. The task of obtaining such sequences is often referred to as preamble generation. Simple search guidance results in a 'flat' cost function landscape and so appears to be insufficient. As our previous work [13][22] did not address this problem, we propose a technique called ***tracing and deducing*** (T&D) in this paper. It helps provide better guidance to the search, allowing our previously developed tools to home in on the targeted test-data in *Simulink*.

2. *SIMULINK* AND TEST COVERAGE
Simulink models are made up of blocks connected by lines. Each block implements some function on its inputs and outputs the results. Outputs of blocks form inputs to other blocks. Models can be hierarchical. Each block can be a subsystem comprising other blocks and lines. This feature allows *Simulink* to handle complexity. Figure 1 is a simple *Simulink* model. It calculates if an equation of form $ax^2 + bx + c = 0$ is a quadratic equation and if it has real-valued solution (s). If it is a quadratic equation and it has one or two real-valued roots, '`1`' is output; otherwise, '`-1`' is output. Block '`IN-A`', '`IN-B`' and '`IN-C`' are three input blocks, receiving the input values of '*a*', '*b*' and '*c*' from the user. Block '`Out`' provides the output.

A `Switch` block is a commonly used for implementing branching in *Simulink*. There is a control parameter 'threshold' associated with each `Switch` block. If the signal carried on the second input port of the `Switch` block 'Vp' satisfies 'Vp ≥ threshold' then input 1 is selected to output. Otherwise, input 3 is selected.

In *Simulink* all blocks execute at each time step. Thus, the traditional code-level concept of 'reaching' a block (i.e. causing it to execute) does not really occur. However, some blocks have conditional behaviours, which is analogous to the behaviours caused by branches in code. Therefore, analogous code-level structural coverage [15] can be defined. In this paper, we adopt the *Branch Coverage* definition by *Reactis Tester* [26]. The ***Branch Coverage*** criterion requires all conditional behaviours of blocks, provided the

block has conditional behaviours, to be executed at least once. For example, a logical operator has two conditional behaviours: 'TRUE' and 'FALSE'. Branch coverage requires that each behaviour be exhibited by at least one test execution. In the current work, we consider three types of blocks that are most widely and often used in forming branches; they are `Switch`, `LogicalOperator`, and `RelationalOperator`.

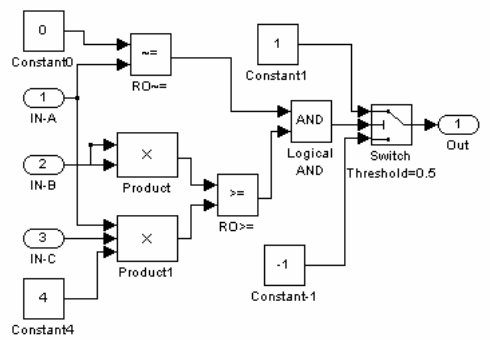

Figure 1. Simple Simulink model.

Simulink is generally used for designing embedded systems – of which a significant feature is that they maintain state. The systems have continuous inputs and outputs and the execution step is controlled by some timer trigger, e.g. a step size can be 1 millisecond. Therefore, for the model in Figure 1, the input to the system over *n* time steps should be a sequence <(IN-A_1, IN-B_1, IN-C_1),(IN-A_2, IN-B_2, IN-C_2),...(IN-A_n, IN-B_n, IN-C_n)>, and the corresponding (single) output should also be a sequence <Out_1,Out_2,...Out_n>. A structural element is considered to be covered if it is covered by any step of the execution.

3. THE STATE PROBLEM

3.1 Automatic Test Data Generation

Test-data generation can be dynamic or static, depending on whether the execution of the test object is involved or not.

The static approach statically determines conditions (in the case of path coverage, the conditions would be path-traversal conditions) that need to be satisfied for the underlying test-data generation aim to be met, and then uses various means (e.g. linear programming, constraint solving) to derive desired test data from them. The constraints are typically generated using symbolic execution [27][28]. The technique, however, has not seen widespread application due to technical difficulties in handling certain language features, such as loops, arrays, pointers and memory allocation.

The dynamic approach executes the software under test. With the guidance information obtained from dynamically running or simulating the underlying test objects, it searches the input domains of the test objects for targeted test-data. This approach has been widely applied in structural testing [1][2][3][4][5] as well as functional testing [6][7][8] and non-functional testing (which has largely focused on temporal testing) [10][11]. (The above works are all carried out at the code level.) Jones et al. [12] have attempted test-data generation from Z specifications. We [13][22] applied the search-based approach to the generation of test-data achieving particular structural coverage or mutation adequacy measures of *Simulink* architectural models.

In search-based testing approaches the satisfaction of a particular test requirement is couched as a sequence of one or more predicates over the behavior of the system before, during, or after execution. For example, a specific path will be taken when the corresponding set of branch conditions hold true during execution [13]. If `X` is in the range [`0..25`], then an exception may be generated at a specific assignment statement `X=Y×Y`, when the healthiness precondition `Y×Y<=25` does not hold before execution of the statement [7]. A more detailed way of specifying the overflow of `X` might consist of a sequence of predicates defining a path that reaches the statement (with no exceptions along the way) together with `Y×Y>25` immediately before the statement. Causing a program to break its functional specification can be couched as satisfying the precondition before execution and not satisfying the post-condition at the end of execution [6].

In order to provide guidance to the search, we must be able to evaluate how close a program execution comes to satisfying a predicate. E.g. for a predicate `X>=50`, a value of `49` for `X` would be considered 'closer' than would a value of `20`. A typical cost function encoding can be found in [22].

Search based testing combines the costs of satisfying various relevant predicates to provide an overall cost for a particular execution. (We omit details here). The aim is to reduce the overall cost to zero. The test-data generation problem becomes a cost function minimization problem; a host of optimization techniques have been adopted, e.g. simulated annealing [6], genetic algorithms [4][5], tabu search [20], and ant colony optimization [24]. Details of search techniques can be found in [16]. The dynamic test-data search is not guaranteed to succeed. Clearly, a search will fail to find appropriate test data when no such data exists (i.e. we are trying to satisfy an infeasible requirement). It may also fail, even when test data actually exists, simply due to the particular strategy employed by the search method. A full account of test-data generation by heuristic search can be found in the McMinn's extensive survey [19].

Combining static and dynamic approaches seems a promising avenue to explore. Offutt et al. [29] have proposed the dynamic domain reduction procedure (DDR) technique, in which run-time information is explored to reduce the domains in an underlying test-data constraint problem. (Our approach, by comparison, uses static techniques to create a more navigable dynamic search problem.)

3.2 The State Problem

Despite the successful use of search-based test-data generation, certain features of systems can hinder the test-data search, e.g. the flag problem [23], and the ***state problem*** [24]. Embedded systems, such as engine controllers, typically make extensive use of state to record real-time information. Such systems usually require test data to be sequences of inputs that can put the system into a certain state, in order to exercise particular elements. Thus, the generation of such input sequences becomes difficult. Usually a coarse objective landscape is yielded and the test-data search easily gets stuck.

The Sort-Code-Verification problem used in McMinn's PhD thesis [25] is an example. The description is as follows:

The system validates a UK bank sort code of the form 'XX – XX – XX' where 'X' is an integer digit. A line feed character is also expected at the end of the input. Unicode characters are submitted to the system one at a time. The system needs to keep track of how far through the validation process the system is. One of three constant integer values are returned. RESULT_ENTER_NEW_CHAR signals the system is ready to accept a new character; RESULT_VALID signals the previously

entered sequence is valid and that the system is ready to read in another sort code; or RESULT_INVALID, which signals that the last character was invalid.

The code description of the problem can be found in [25]. Figure 10 in section 8 is its *Simulink* version. Characters are read from `In1` (in section A) at each time step. The ASCII encodings of characters are used. Section B contains components that determine when characters of these types have been entered. The parse of a sort code proceeds by successively reading its constituent characters. The system may be in one of several state positions: `1` to `9`. The position is incremented by `1 (in section C)` as the parser reads successive characters in a valid sort code sequence supplied at the input. Section D checks what position the state is in (and so what sort of character is expected). Section E determines whether the current input is of the form expected. If an invalid character is input (the information is shown as the output of the `NOT` block above section E), the system will reset to position 1. Section F determines the output signal for the whole system.

In the model, the states are maintained by the `UnitDelay` block passing the information in the previous step back into the system for use by the new step. Assume that our test goal is to cause the output of block 'pos9+lf' to be TRUE. This requires a valid sort code input. To satisfy such a test aim manually we can derive the following constraints: the targeted test datum should consist of at least `9` steps; and the shortest sequential input should be: digit, digit, dash, digit, digit, dash, digit, digit, and line-feed. Standard cost function design as described previously [13][22] will usually generate two constraints: the value of 'In1' equals to `12` and the runtime value of the output of block 'pos' equals to `9`. Such constraints do not tell the minimum number of steps needed to execute. The cost function landscape of the second constraint will be `9` plateaus; each indicates one of the states of the system. Therefore the search will obtain little guidance from these constraints and result in failure. The kind of guidance the search really needs is something like the constraints that are derived manually.

We therefore will introduce a technique called ***tracing and deducing*** (T&D), which uses more detailed constraint information to replace the rough guidance. The more detailed constraint information allows a more easily navigable landscape to be created. Such a process enables existing search tools to home in on targeted test-data.

4. TRACING AND DEDUCING

Suppose that some testing goal is given. This would typically be a requirement to satisfy some predicate `P` over line (signal) values at a certain step in the execution of the system, e.g. a branch condition. For simplicity assume that this is a predicate over a single signal. This signal will be the output of some block `B`. By back-propagating the signal through block `B`, the predicate can be replaced by a new predicate or a conjunction or disjunction of multiple predicates. The new predicate(s) is (are) over the input signal(s) of block `B`. We call a replacement like this one 'application' of the deducing process.

When a goal is defined over the output of a `UnitDelay` block, a refined goal defined over its input signal value at the previous time step can be derived. By tracing back recursively in this way, more helpful guidance to the targeted test-data search can be derived. This is in fact a kind of reverse symbolic execution. No *complete* symbolic execution is required. Certain stopping rules are applied for the process so that the problems of standard symbolic execution do not apply. A simplification process is also used with the tracing and deducing process in order to keep the complexity of predicates under control.

4.1 Assumptions

Our approach requires us to identify a suitable number of steps over which to consider the execution of the system. It is assumed that the goal will be met on the final step. Section 4.6 will describe how to obtain this 'suitable number of steps.'

A branch-coverage testing goal can be interpreted into a predicate defining the value range of a particular signal value at the final step of the execution. For example, if the goal is to have the output of block '`pos3+dash`' to be '`TRUE`', the goal predicate will be: the output value of block 'pos3+dash' on the **N**th step is 'TRUE' (`N` is the minimum step size identified by the technique described in section 4.6). Starting from this initial predicate (constraint), new constraint(s) can be deduced. All constraints should concern values of signals at a certain time unit. Each signal has a unique integer label (determined by its source block name and source port number). The following notation is used in denoting constraints. The value of the 5^{th} step of signal `32` will be denoted as '$P_{32}(5)$'. '`P`' represents term 'Probe', which indicates that we have to insert a probe in to that signal to detect its value. In a similar way, a constant value of `58` will be denoted as '`C(58)`', where '`C`' represents 'Constant'.

Constraints are always in the form of a relational predicate or various logical combinations of relational predicates, such as:

$P_{32}(5)>=C(58);$

$\{P_{32}(5)>=C(58) \vee P_{32}(5)<=C(47)\} \wedge P_{32}(5) \neq C(12).$

The constraints are recorded in a tree-like structure, which is composed of two types of nodes and lines. The tree structure is called an ***objective-tree***. The data structure of the tree-like graph will be described in section 4.2.

The *T&D* process is really a process of constructing the *objective-tree*, starting from one single node, which is the goal predicate. The construction process involves tree node refinement and tree simplification.

4.2 Storage Rules

There are two types of nodes in the *objective-tree*: ***Predicate-Node*** and ***Or-Node***. A *Predicate-Node* records the information of an atomic constraint, which is a relational predicate, and its 'next' domain, which is a pointer pointing to the node that has a conjunctive 'AND' relation with it. An *Or-Node* can have a number of children; each is denoted by a pointer to the corresponding *Child-Node*, which will, in turn, be a *Predicate-Node* or an *Or-Node*. The relation between the children of an *Or-Node* is disjunctive 'OR'. An *Or-Node* also has a 'next' domain, pointing to the node that has an 'AND' relation with all its children. If the 'next' domain of a node is '`0`', it means no more predicates will be included in this conjunctive relation. Therefore, in the objective-tree representation, branches represent disjunctions; linear sequences represent conjunctions. In an *objective-tree*, an *Or-Node* is denoted by a triangle, with arrows coming out of its left corner representing the child pointers and one arrow originating from the middle of its bottom side representing the next pointer; a *Predicate-Node* is denoted by a rectangle with one arrow originating from the middle

of its bottom side representing the next pointer. For both types of nodes, the next pointer may or may not exist.

For example, predicate $\{P_{32}(5)>=C(58) \vee P_{32}(5)<=C(47)\} \wedge P_{32}(5) \neq C(12)$ can be denoted as shown in Figure 2. In the figure, the numerals, such as 1, 2, 3, 4, are node numbers. Node 1 is an *Or-Node*, having two children – node 3 and node 4. The 'next' domain of node 1 points to node 2.

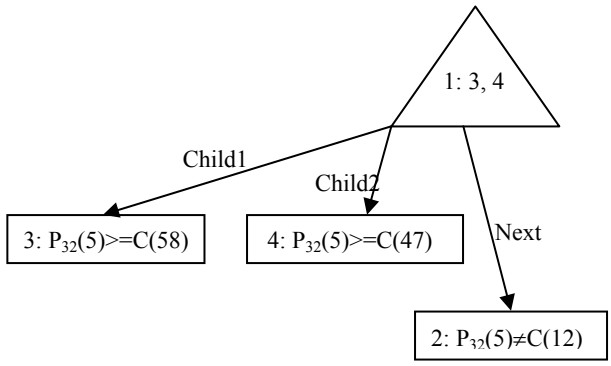

Figure 2. An example of objective-tree representation.

4.3 Deducing Rules

The deducing activity is the process of refining predicates. It is called on by the tracing process, as described in section 4.4.

If a predicate is deduced, a new predicate or a few new predicates will be generated to replace the original one. Some deduction rules for various types of *Predicate-Nodes* are defined below. These rules have been implemented into a prototype tool. However, the deduction rules may not be restricted to these. More rules about deduction for other types of blocks can be added to the set.

In this prototyping tool implementation, for simplicity, a predicate is not deduced if both of its operands are probes.

A predicate can be deduced to 'TRUE' or 'FALSE' when both of its operands are constants. Then the *objective-tree* can be simplified accordingly (see section 4.5 Simplification Rules).

For a predicate of form '$P_x(y)$ *rel* `C(z)`', it can be deduced when the source block of probe 'x' is: `Switch`, or `LogicalOperator`, or `RelationalOperator`, or `UnitDelay`, or `Sum` and it has only one non-constant input, or `Product` and it has only one non-constant input.

Below we describe how each of these blocks can be deduced.

4.3.1 'Switch' Block

If the source block is `Switch`, as illustrated in Figure 3:

According to the functionality of the `Switch` block (given in section 2), there are two ways the predicate '$P_x(y)$ *rel* `C(z)`' can be satisfied: the second input of `Switch` is greater than or equal to the 'Threshold' parameter and the first input satisfies the predicate requirement as `x` does; or, the second input of `Switch` is less than the 'Threshold' parameter and the third input satisfies the predicate requirement as `x` does. The step number of the signals in the newly deduced constraints should be the same as in the original predicate. The implementation is as described below.

The original *Predicate-Node* '$P_x(y)$ *rel* `C(z)`' will be changed into an *Or-Node*, with two children – `newNode1` and `newNode2`. Its 'next' domain remains the same.

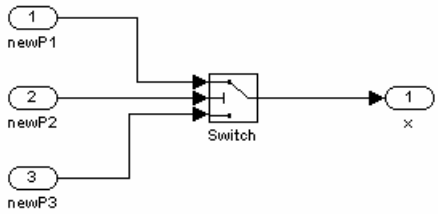

Figure 3. Deductive process for Switch block.

Four new nodes are created: `newNode1`, `newNode2`, `newNode3` and `newNode4` (see Figure 4). The 'Threshold' ('`thres`') of the `Switch` block will be detected.

`NewNode1` will be recorded as predicate '$P_{newP2}(y)$ `>= C(thres)`'. Its 'next' domain will be '`newNode3`'.

`NewNode2` will be recorded as predicate '$P_{newP2}(y)$ `< C(thres)`'. Its 'next' domain will be '`newNode4`'.

`NewNode3` will be recorded as predicate '$P_{newP1}(y)$ *rel* `C(z)`'. Its 'next' domain will be '`0`'.

`NewNode4` will be recorded as predicate '$P_{newP3}(y)$ *rel* `C(z)`'. Its 'next' domain will be '`0`'.

The new tree structure is:

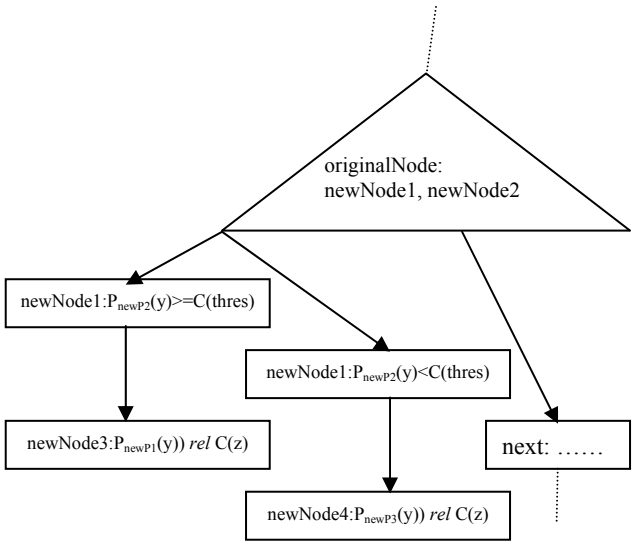

Figure 4 Tree node after deducing a Switch block.

4.3.2 'LogicalOperator' Block

If the source block of signal `x` is `LogicalOperator` or `RelationalOperator`, since the output of such blocks must be either '`0`' or '`1`', the original predicate node '$P_x(y)$ *rel* `C(z)`' needs to be interpreted. For example, if *rel* is '>' and `z` is '`0.5`', the predicate will be interpreted into '$P_x(y)$ `== C(1)`'; if *rel* is '<' and `z` is '`0.3`', the predicate will be interpreted into '$P_x(y)$ `== C(0)`'; if *rel* is '<' and `z` is '`0`', the predicate will be deduced to '`FALSE`' since that is impossible; if *rel* is '≠' and `z` is '`5`', the predicate will be deduced to '`TRUE`' since it is always

true. A value of '0' or '1' of $P_x(y)$ is called the ***target result*** of such source blocks.

If the logical operator is 'AND':

If the *target result* is '1', in order to satisfy this predicate, all inputs of the block need to be TRUE. If the *target result* is '0', in order to satisfy this predicate, at least one of the inputs of the block needs to be FALSE.

If the logical operator is 'OR':

If the *target result* is '1', in order to satisfy this predicate, at least one of the inputs of the block needs to be TRUE. If the *target result* is '0', in order to satisfy this predicate all inputs of the block need to be FALSE.

If the logical operator is 'NOT', as illustrated in Figure 5, In order to satisfy the predicate, the value of the input signal of the block should be exactly the opposite of the *target result*.

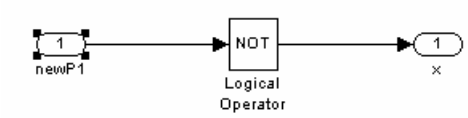

Figure 5. Deductive process for 'NOT' Logic block.

Detailed implementation will be similar to the treatment of a Switch block and is omitted here.

4.3.3 'RelationalOperator' Block

If the source block is RelationalOperator (like in Figure 6), as explained in section 4.3.2, the original *Predicate-Node* '$P_x(y)$ *rel* C(z)' can be interpreted into one of the following two forms: '$P_x(y)$ == C(1)' or '$P_x(y)$ == C(0)'. To satisfy such a predicate, the inputs of the block should satisfy the relation defined by the operator in the first case, or fail to do so in the second case.

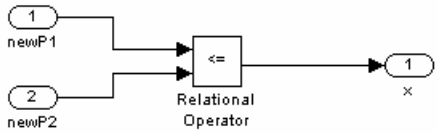

Figure 6. Deductive process for RelationalOperator block.

4.3.4 'UnitDelay' Block

If the source block is UnitDelay, as in Figure 7, according to the functionality of the UnitDelay block (the output of it equals the value of its input signal in the previous step), the input signal of the UnitDelay block in the previous step should satisfy exactly the requirement for the output signal of the UnitDelay block defined by the original predicate. The implementation can be defined accordingly:

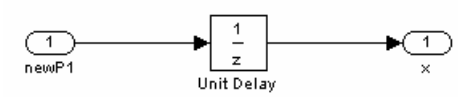

Figure 7. Deductive process for UnitDelay block.

If the step number y equals to '1', the original *Predicate-Node* '$P_x(y)$ *rel* C(z)' will be deduced to 'TRUE' or 'FALSE' accordingly. This is because the output value of a UnitDelay block is always '0' for the first step. Otherwise, the original *Predicate-Node* will be changed into '$P_{newP1}(y-1)$ *rel* C(z)'. Its 'next' domain remains the same.

4.3.5 'Sum' or 'Product' Block

If the source block is Sum or Product and it has only one non-constant input, the predicate can be deduced into a constraint about the non-constant signal. By example, for the situation illustrated in Figure 8, if the constant is A, the original *Predicate-Node* '$P_x(y)$ *rel* C(z)' will be changed into '$P_{newP1}(y)$ *rel* C(z-A)'. Its 'next' domain remains the same.

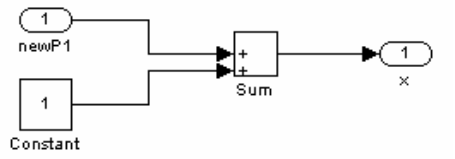

Figure 8. Deductive process for Sum block.

4.4 Tracing Stopping Rules

The tracing process is a recursive process. In every pass, it checks each available node in turn; if deducible, the deducing function is called upon. Simplification (introduced in the next sub-section) to the *objective-tree* occurs at the end of each pass.

A node will not be further traced and deduced if both of its relational operands are probes.

Since the output value of a UnitDelay block is always '0' for the first step, a node cannot be further traced when its step number is '1' and the probe's source block is UnitDelay.

When a signal is about to be traced through a UnitDelay block, which means the signal step number will be reduced by '1', the program will withhold the tracing back until the tracing back process of all the other signals reaches the same situation. This rule assists the synchronization of the tracing back process and enables significant simplification of the constraints earlier in the *objective-tree* construction procedure.

To avoid the typical problem caused by symbolic execution (i.e. *objective-tree* explosion), a maximum tree-size is imposed in the tracing process. The process is stopped when the number of nodes reaches this limit. This does not prevent us using heuristic search to find the ultimate test-data, but the guidance provided to the search may be less informative.

4.5 Simplification Rules

During the *tracing and deducing* process the *objective-tree* gets bigger. The tree is repeatedly simplified.

There are a few simplification treatments defined in the prototyping tool. For example, if a node is determined to be constantly TRUE, the node will be deleted (as illustrated in Figure 9); if a node is determined to be constantly FALSE, the branch the node belongs to will be removed. Simplification methods are also defined for cases such as: an *Or-Node* has only one child branch left or no child branch at all (such a situation results from previous node-TRUE or node-FALSE simplification), two nodes on one path are conflicting, and two nodes on one path are consistent (i.e. the satisfaction of one can assure the satisfaction of the other). Due to limitations of paper length, details of the implementation of the simplification rules are omitted here. Interested readers can refer to [30].

If a tree is simplified to be empty, it indicates that the test-data generation is infeasible. One reason might be the number of steps is

too small and needs to be augmented. (This will be discussed in more detail in section 4.6.)

The current simplification tool is only a proof-of-concept implementation. Complicated conflicting constraints cannot be identified nor thereafter be simplified. For example, constraint `(A<=0.3∨B>=3)∧ (A>0.3∧B==2)` cannot be identified as `FALSE`. The usefulness of the T&D approach may be enhanced by incorporating more powerful constraint solving tools.

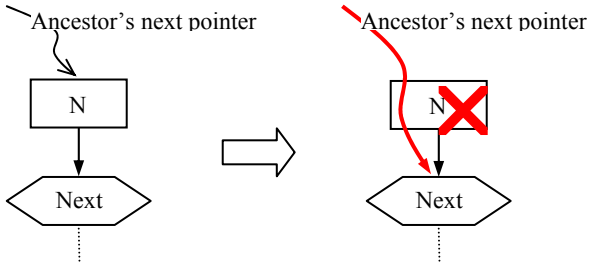

Figure 9 Simplification for Node-True.

4.6 Discussions

According to the first assumption in section 4.1, to use T&D the number of steps to execute needs to be given. When the number given is smaller than the minimum number of steps required to satisfy the test-data generation requirement, the T&D process may fail (result in an empty *objective-tree*). If the number given is larger than necessary, it usually results in complicated constraints being deduced due to the variability involved. A strategy of trying different numbers from small to large, one by one, is suggested, in searching for the appropriate number of steps to execute. When the number tried is too small, the T&D program usually fails quickly (tells that the constraints are not satisfiable). This approach avoids giving too much flexibility in the constraints and ending up producing a large, complicated, and potentially unworkable *objective-tree*.

The technique can be considered as a partial symbolic execution. It is partial because it does not fully execute the system. It executes only to the extent that the symbolic conditions can be handled without any difficulty. This is controlled by the tracing stopping rules. The technique allows some of the benefits of symbolic execution to be obtained within the context of a heuristic search approach, but without suffering its state explosion problems.

5. RESULTS

An experiment was run with a preliminary implementation of the *T&D* technique. *T&D* was implemented as an additional component that we can insert into our previously implemented standard search-based test-data generation tool for *Simulink*. The search engine applied is *Simulated Annealling* (SA) [18]. In our application a move effectively perturbs the value of one of the inputs of one step in the current test sequence by a value less than or equal to 2 percent of the range of the input. We applied a geometric cooling rate of 0.8. The number of attempted moves at each temperature was 100, with a maximum of 300 temperature reductions. These parameters may be thought to be on the 'small' side, but the computational expense of simulation necessitated pragmatic choices.

In this experiment, five models are tested; four test-data generation approaches are compared. The five models are: Sort-Code-Verification, Post-Code-Verification, Smoke-Detector, Inputs-Check, Sys-Fuel-Dip-Ign-Req. The first three problems are borrowed from McMinn's PhD thesis [25]. The fourth one was created by the author. The fifth, an engine controller subsystem, was taken from industry. In this experiment, the input ranges of the first two problems are modified to `[1 .. 1000]`, which is smaller than the ranges used by [25] (`[0 .. 65535]`). The modified setting helps to better demonstrate the different capabilities of various test-data generation techniques.

The four test-data generation approaches are: random testing, standard SA search-based testing, *tracing and deducing* technique facilitated SA search-based testing and *Reactis Tester* (as mentioned in section 2). They are named RAND, SA-STD, SA-T&D and Reactis respectively. The aim of this experiment is to compare the performances of various test generation approaches. When the RAND and SA-STD approaches (which do not have self-adjustment ability to set the sequences lengths) are used, it is ensured that the fixed sequence lengths for generating the appropriate test-data were long enough for all targets (i.e. branches in this experiment) to be covered.

The experiment was performed on a [Intel Pentium M 1.6 GHz, 512M RAM] laptop computer. It is made up of two parts. The first part examines the total coverage achievements of each approach. The goal is to achieve all *branch coverage*, as defined in section 2. For each branch coverage goal, the SA-STD and SA-T&D methods, according to the SA cooling schedule setting, the maximum number of attempts to achieve the coverage of one branch will be `30,100` (of which `100` is used in determining the initial temperature). For RAND and Reactis, each approach is allowed `100,000` attempts (evaluations of test-data) to achieve an individual branch coverage goal[1]. Table 1 shows the results. The aim of this part of the experiment is to show that the T&D technique enhances the solvability of the SA-STD approach (i.e. can achieve higher coverage). The second part of the experiment compares the time cost of different approaches in generating an input sequence to cover an individual branch. Again each approach is allowed of `100,000` attempts. Since *Reactis Tester* cannot be set to target one individual branch at a time it is not included in this comparison. Table 2 illustrates the results. The aim of this part of experiment is to show that with the aid of the T&D method, the SA-T&D approach outperforms the SA-STD approach, in terms of solving success rate and time cost, for some individual branch coverage requirements.

All results in Table 1 and Table 2 are based on the average of `10` individual runs. Some of the runs may be unsuccessful, given the limit of attempts imposed. So, it is reasonable to assume that the real time cost is larger than our cost (hence the prefix of '>' in Table 2).

Table 1 Average coverage achievement comparison

Model	No. of Branches	RAND	SA-STD	SA-T&D	Reactis
Sort-Code-V.	56	76.4%	86.8%	100%	75%
Post-Code-V.	76	84.2%	91.7%	100%	84.2%
Smoke-Detector	34	88.2%	90.0%	94.1%	88.2%
Inputs-Check	8	75%	87.5%	100%	100%
Sys-Fuel-Dip.	20	65%	93%	82%	65%

[1] Since Reactis attempts to cover all branches at a time (no individual branch can be targeted), it is allowed (`number_of_branches × 100,000`) attempts each time to cover a whole model.

Table 2 Time cost (seconds) comparison

Prob. No.	RAND	SA-STD	SA-T&D	
			Total	Tree Construct
1	>6347 (all fail)	712	141	30.5%
2	>12101 (all fail)	1386	133	58.6%
3	>2017 (all fail)	>603 (70% fail)	839	88.2%
4	>2457 (all fail)	>789 (50% fail)	54	3.1%
5	>2714 (all fail)	>287 (20% fail)	>323 (40% fail)	<1.2%

Taking the whole Table 1 into account, the SA-T&D approach demonstrates superiority in covering models by and large.

Table 2 details the results of the second part of the experiment. For each of the five models identified in Table 1, a specific difficult branch coverage requirement was subjected to experiment. (Problem 1 corresponds to a difficult branch of Sort-Code-Verification, problem 2 corresponds to a difficult branch of Post-Cod-Verification, and so on.) The ratio of time cost for objective-tree construction (incurred by the T&D algorithm) is also given as reference information for readers. As can be seen, for the third problem, the majority of the cost was spent in the tree construction rather than SA search. Such high cost is due to the limitation of the prototype implementation of the simplification tool.

The T&D approach failed to achieve full coverage for two models in the experiments. For Smoke-Detector, the failure was also due to the immature constraint simplification technique used in the prototype tool. Advanced constraint solving tool should solve the problem. For Sys-Fuel-Dip-Ign-Req, the inability to achieve full coverage was partly due to infeasibility of one branch. In comparison to the SA-STD approach, the SA-T&D approach has a lower success rate in covering Sys-Fuel-Dip-Ign-Req. Detailed information shows that the difference lies in the different solving success rates of 5 of the branches. For this particular problem, SA-STD was allowed 3 steps for each test-datum evaluation whilst the minimum requirement is 2 steps (which is the sequence length used by SA-T&D). Therefore, in each test-datum evaluation, SA-STD actually had two opportunities (goal achieved in the 2nd or 3rd step) to win the assessment while SA-T&D had only one (goal achieved in the 2nd step). This is also the reason SA-T&D costs more than SA-STD in the fifth problem of Table 2.

As mentioned earlier, we reduced the input range of problem Sort-Code-Verification and Post-Code-Verification in order to show discriminating solvability of different approaches. We have also tried to use the SA-T&D method on the difficult versions of the problems as used in [25]. Full coverage was also achieved. Zhan's thesis [30] gives evidence.

6. CONCLUSIONS

The results have shown that the *tracing and deducing* technique can substantially enhance the capability of search-based test data generation for *Simulink*. It improves both the time to produce test data and the coverage.

The *T&D* technique proposed here is actually a partial symbolic execution procedure. It attempts to exploit the capabilities of symbolic execution in order to provide more informative guidance to the search whilst avoiding the disadvantages of symbolic execution by imposing certain stopping rules to it. Such a concept should apply to code level testing too. However, the 'tracing' process for code will not be as simple as in *Simulink*. It should rely on some kind of semantic analysis.

The *T&D* technique is not restricted to search-based test-data generation. Taken some tracing stopping rules away, it should apply to test data generation through other techniques such as constraint solving (which is employed by *Reactis Tester*), etc..

There has been considerable success applying heuristic search techniques for code level test data generation. The work presented in this paper forms part of a larger investigation into the application of search techniques to test-data generation for *Simulink* models. A framework has been created that applies search-based techniques to structural coverage problems [13] and mutation coverage problems [22]. In addition, test set optimization is also included. The full framework, including the tracing and deducing work, is described in [30]. The work is very much in keeping with the current emphasis on "model-based testing". The state problem remains a challenge for higher level as well as code level test-data generation. Automatic generation of test data for higher level models more generally is a very challenging area. We recommend both as promising research areas to the Search-Based Software Engineering community.

7. ACKNOWLEDGMENTS

Our thanks to Rolls-Royce for sponsoring this research.

8. REFERENCES

[1] B. Korel. Automated Software Test Data Generation. *IEEE Trans. on Softw. Engineering, 16(8): 870-879*, 1990.

[2] N. Tracey, J. Clark, K. Mander, and J. McDermid. An Automated Framework for Structural Test-Data Generation. *Int'l Conf. on Auto. Softw. Eng.,* pp 285-288, 1998.

[3] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. *GECCO 2002,* pp 1233-1240.

[4] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gal, S. Katsikas and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Int'l Conf. on Softw. Engineering and its Applications,* pp 625-636, 1992.

[5] B. Jones, H. Sthamer, and D. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal,* 11(5): 299-306, 1996.

[6] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. *Symposium on Software Testing and Analysis (ISSTA),* pp 73-81. 1998.

[7] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated Test Data Generation for Exception Conditions. *Software – Practice and Experience, 30(1): 61-79*, 2000.

[8] O. Buehler and J. Wegener. Evolutionary Functional Testing of an Automated Parking System. In *Int'l Conf. on Computer, Communication and Control Technologies (CCCT'03) and The 9th Int'l Conf. on Information Systems Analysis and Synthesis, (ISAS'03)*, 2003.

[9] A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms. *GECCO 2003, pp 2428-2441.*

[10] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer and B. Jones. Systematic Testing of Real-Time Systems. *Proc. of the*

4th European Conference on Software Testing, Analysis & Review (EuroSTAR '1996), Dec. 1996.
[11] P. Puschner and R. Nossal. Testing the Results of Static Worst-Case Execution-Time Analysis. *Proc. of the 19th IEEE Real-Time Systems Symposium, pp 134-143*, 1998.
[12] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The Automatic Generation of Software Test Data Sets Using Adaptive Search Techniques. T*he 3rd Int'l Conf. on Software Quality Management, pp 435-444*, 1995.
[13] Y. Zhan, and J. Clark. Search-Based Automatic Test-Data Generation at an Architectural Level. *GECCO 2004, pp 1413-1426*.
[14] Leonardo Bottaci. Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data. *GECCO 2003,* pp 2455-2464.
[15] Hong Zhu, Patrick A. V. Hall and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys, Vol. 29(4): 366-427*. December 1997.
[16] C. R. Reeves (Ed.). *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell, Oxford, 1993.
[17] R. Kirner, R. Lang, G. Freiberger and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. *Euromicro Conference on Real-Time Systems*, 2002.
[18] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science, 220(4598): 671-680*, 1983.
[19] P. McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability, 14(2), pp 105-156*, June 2004.
[20] Eugenia Díaz, Javier Tuya, Raquel Blanco. Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. In *18th IEEE Int'l. Conf. on Automated Software Engineering*. Montreal, Canada, Oct. 2003.
[21] The MathWorks. http://www.mathworks.com/products/simulink.
[22] Y. Zhan, and J. Clark. Search-Based Mutation Testing for Simulink Models. *GECCO 2005*, pp 1061-1068.
[23] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving Evolutionary Testing by Flag Removal. *GECCO 2002, pp 1359-1366*.
[24] P. McMinn, and M. Holcombe. The State Problem for Evolutionary Testing. *GECCO 2003,* pp 2488-2500.
[25] P. McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD Thesis, University of Sheffield, January 2005.
[26] Reactive Systems Inc. Http://www.reactive-systems.com/.
[27] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering, 2(3)*: 215-222. 1976.
[28] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering, 17(9)*: 900-909. 1991.
[29] A. J. Offutt, Z. Jin and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. Software – Practice and Experience, 29(2): 167-193. 1999.
[30] Y. Zhan. *A Search-Based Framework for Automatic Test-Set Generation for MATLAB/Simulink Models*. PhD thesis, University of York. Dec 2005.

9. APPENDIX: 'sortCodeVerificaiton'

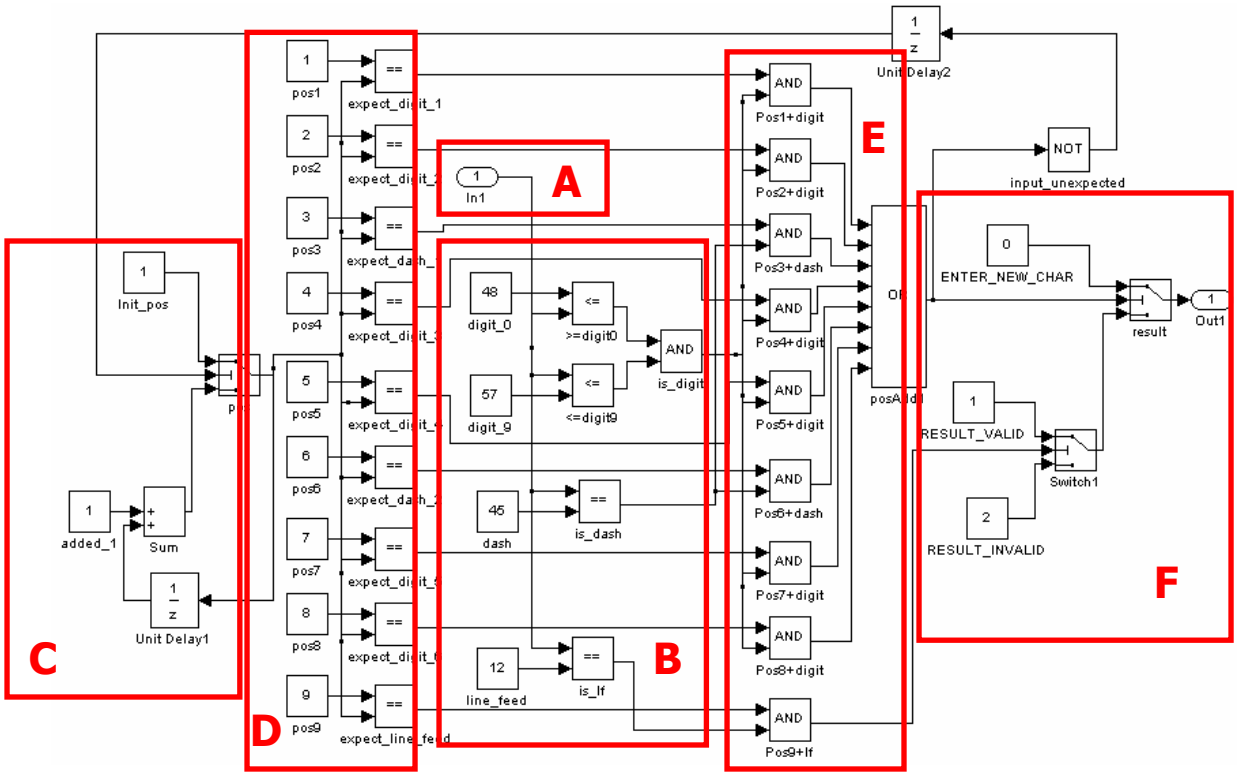

Figure 10. Simulink model of 'SortCodeVerificaiton'.