Automatic Test Generation for Model-Based Code Generators

Sergey V. Zelenov[1], Denis V. Silakov[1], Alexander K. Petrenko[1], Mirko Conrad[2], Ines Fey[3]

[1] ISPRAS, Russian Federation, {zelenov | silakov | petrenko}@ispras.ru

[2] Member of the ACM, mirko.conrad@acm.org

[3] DaimlerChrysler AG, Germany, ines.fey@daimlerchrysler.com

Abstract— **This paper presents a novel testing approach for model-based design tools, termed GraphOTK, and applies it to the optimizing component of a code generator for Simulink and Stateflow models.**

Index Terms—**code generation, model-based development, Simulink, test generation**

I. Introduction

The increasing complexity of software systems requires facilities that allow developers to think directly in terms of a subject domain during the development process.

However, modern software applications are so complicated that even the use of modern textual programming languages does not allow the developer to observe the entire system as a whole. Quite often the implementation of requirements that are obvious from the point of view of a subject domain and that can be formulated by one sentence leads to the creation of hundreds of lines of program code.

A very promising approach to manage system complexity is the Model-Based Development (MBD). This technology implies the use of domain-specific modelling languages (DSML) that allow the design ideas to be expressed in a more declarative form. The technology is supported by a wide range of tools for instance for automatic checking and analysis of developed models, for creation of various kinds of artefacts, such as executable program codes or project documentation, and for different model transformations.

Specific for MBD realizations is the graphical modelling using block diagrams and state charts that is now widely applied in different industry fields: automotive (Ford, General Motors [5] Daimler Chrysler [6]), aerospace (Boeing [7], Honeywell [4]) and others.

One of the most widespread application fields of graphical modelling is the embedded systems development. Modelling and simulation are used during all stages of embedded system development. The program code itself is obtained from software models using so-called *code generators*.

Manuscript received June 14, 2006.

The application of graphical modelling in embedded system development has the following advantages:

- Simplicity of model design process for developers (no formal model description language knowledge is required);
- Automatic generation of system specification and documentation on the basis of the model (in fact, the model itself is an illustrative and at the same time formal specification of the system);
- Automatic code generation for embedded systems;
- Moreover, modern tools for model design allow creation of very complicated models that consist of large amounts of small modules.

Model-based development with graphical languages is supported by commercial modelling tools, such as the Simulink product family from The MathWorks [1], ASCET-SD by ETAS [2] and I-Logix' Statemate [3] that combines traditional graphical design notations with some of the UML diagrams. Tools such as Real-Time Workshop Embedded Coder by The MathWorks [1] are used for code generation from Simulink models. Also, there are different code generation tools, which can be used for instance for ASCET-SD and Statemate models. Nowadays, the Simulink product family seems to be the de-facto standard within the automotive powertrain area. It is used by DaimlerChrysler, General Motors, Ford, Toyota [5], and others.

Code generators are widely used for automotive control software development, including an increasing degree of safety-critical applications. Therefore, code generators have to meet stringent requirements.

There are two primary groups of requirements:

- *Correctness requirement*: The code structure must comply with the model;
- *Efficiency requirement*: The code generated should be of the same quality as the manually written code, considering both memory consumption and execution time.

Tools that meet the efficiency requirement were presented not long ago, at the end of the 1990s. In order to generate high-quality code, different optimizing transformations are applied to the models and downstream intermediate representations. Since models in practice are very complex, a large number of different transformations are performed during code generation. The result of these transformations is highly optimized code in terms of memory and timing constraints.

The main problem is that the behaviour of the optimized code will not comply with the behaviour of the model if errors occur during the optimization process. Therefore the semantics of code obtained from the optimized model will not comply with the semantics of the code that would be obtained without optimization, i.e. the correctness requirement will be broken.

The quality of model-based code generators in general, and with respect to the correctness requirement in particular, can be assured and even increased by systematic testing. As mentioned above, the major error potential comes from the optimization algorithms that are part of the code generation process. Considering this, one focus in testing code generators should be on the optimization rules.

In general the most crucial part of the testing process is the test case design, as is the case here. When testing optimization components, test case design means providing test models which should be transformed into code during the test and which activate the execution of specific optimization rules. The situation is aggravated by the fact that optimizers can perform a lot of different transformations – meaning thousands or tens of thousands of tests are required for code generator testing. Therefore it is necessary to automate the test case generation process.

The remainder of the paper is structured as follows: Section 2 summarizes two existing approaches to design test models for testing code generators. Section 3 introduces GraphOTK, a test generation approach for optimizer components of model-based code generators. Section 4 applies the GraphOTK approach to optimizations of Simulink / Stateflow models, and Section 5 summarizes the main ideas.

II. Existing Approaches

Since the industry has begun to use code generators only recently, they are not as mature as, for example, programming language compilers, and there are not many approaches to code generator testing. We can mention two of them: the formal proof of code generator correctness (e.g. [8]) and test generation based on the Classification Tree Method (CTM) [9-11]. For a broader overview, see [12].

For formal definitions of the optimizing transformations performed by code generators, graph transformation rules can be used [6]. Each rule consists of two parts – a left-hand side (LHS) and a right-hand side (RHS). The LHS specifies the search pat-tern for the optimizer that should be transformed. Also logical application conditions can be defined that should be checked before transformation. The RHS specifies the result structure that will be obtained after the transformation.

Formal proofs are quite difficult and expensive. Moreover, the input languages for popular model design tools such as Simulink/Stateflow are typically not formally defined. Nowadays formal proof is not widely used for industrial code generators.

The Classification Tree Method can be successfully used for generating tests for real systems on the basis of their models [10], but the proposed test generation approach for code generators includes manual steps.

The process of creating a classification tree can be divided into two main steps:

• The input domain of a test object is split up into partitions, called *classifications*;

• Each classification is subdivided into equivalence *classes*.

The resulting classifications and classes are graphically depicted by so-called classification-trees.

Fig.1 shows the simplified classification tree for constant folding on arithmetic expressions. The classifications here are const and sum. Equivalence classes are constructed with regard to the number of nodes.

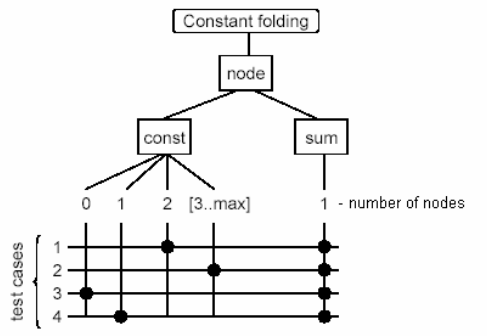


Fig. 1. Classification tree for constant folding

Test cases are then derived manually by selection and combination of equivalence classes, but for real systems the number of equivalence classes can be huge. One of the advantages of the CTM is the simplicity of coverage criterion formulation. To achieve a good coverage for each of these classes, at least one test case should be generated. However, at present, code generator testing can only be performed by proprietary tools [11].

This paper suggests another testing approach, called GraphOTK[1] , that automates the testing of tools which operate on various types of models, such as model-based code generators. GraphOTK solves the problem of automatic test case generation, as only the *abstract generation model*[2] development and the adjustment of the generators are performed manually. Test generators created according to the approach can be adjusted to produce test case sets meeting different coverage requirements (and, therefore, containing different number of tests). The GraphOTK testing method has been applied to optimizers which are part of model-based code generators (short: graphical model optimizers). Test generation is performed on the basis of formal models that describe different pattern structures which should be transformed by the optimizing components.

[1]Graph Optimizer Testing Kit

[2]An *abstract generation model* is a model that consists of essential elements only which must be present in test models.

III. Automatic Test Generation Approach for Model Optimizers Testing

The GraphOTK method is intended for the checking of MBD tools operating with models. This method allows the automatic generation of test cases representing complex graph structures. Such test cases, which are models themselves, can be used to test graphical model optimizers in particular. The method is based on a model-based approach to generate test data for optimizers in programming language compilers [13, 14].

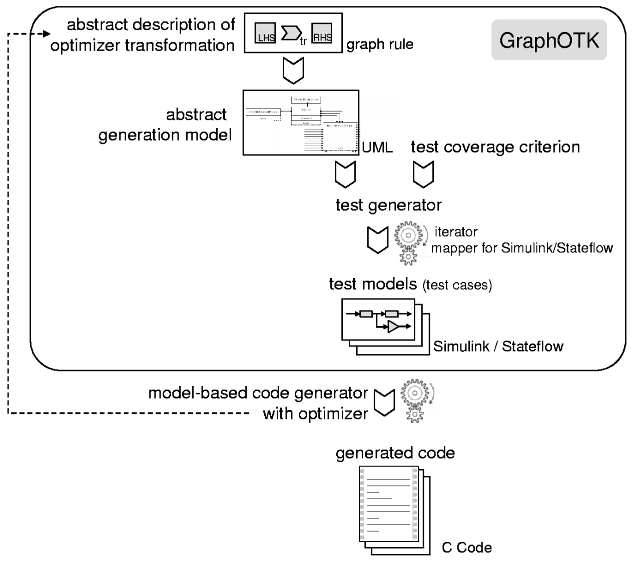

abstract description of optimizer transformation
LHS
tr
RHS
graph rule
GraphOTK
abstract generation model
UML
test coverage criterion
test generator
iterator
mapper for Simulink/Stateflow
test models (test cases)
Simulink / Stateflow
model-based code generator with optimizer
generated code
C Code

Fig. 2. The GraphOTK approach

The GraphOTK method allows testers to obtain the representative set of tests for the system, i.e. tool, under test. The test set creation process involves the following steps (cf. Fig. 2):

1. Creation of an abstract generation model for the test cases: The abstract generation model is a UML model that is constructed on the basis of an abstract description of the model processing algorithm, i.e. optimization rule, to be tested. Such an abstract test generation model allows the tester to concentrate on the most critical aspects of the model processing algorithm and finally to obtain a representative test suite.
2. Definition of a test coverage criterion in terms of the abstract generation model: The test coverage criterion allows the quality of the test suite to be predetermined. It is formulated in terms of parts of the abstract generation model to be covered.
3. Creation of test generators for the generation of test suites that meet the test coverage criterion: Test generators solve the main problem mentioned above – they automate test case generation. The test generator of GraphOTK generates test model structures one by one and then maps them to the required format, which depends on the tool under test. Test generators can be configured in different ways and, in the event of changes to the coverage criterion, a new test suite meeting the modified requirements can be easily obtained by adjusting the corresponding generator settings.

The following paragraphs give more detailed description of the above-mentioned steps with reference to the testing of code generator optimization components.

A. *Creation of an abstract generation model*

An abstract generation model is constructed on the basis of an abstract description of the optimization algorithm.

The optimization algorithm to be tested is described by means of graph transformation rules. In order to perform the transformation, the optimizer searches for entity combinations that match some *patterns*, i.e. combinations of model elements represented in the LHS of the graph transformation rule. In order to construct an abstract generation model, we will consider only those graphical model blocks that are involved in at least one pattern.

On the basis of the information about pattern structures from all graph transformation rules for the optimization under consideration, the set of all model elements that are used in these patterns is constructed. Next, a set of abstract generation model building blocks is described as follows:

- Each element of the model corresponds to a certain kind of building block of the generation model;
- Building blocks may be connected in order to form structures, which correspond to patterns.

Let *test model structure* denote a graph with building blocks as vertices and links between building blocks as edges.

B. *Definition of a test coverage criterion*

The projection of the graphical models on the set of model structures induces a partitioning of all possible graphical models into equivalence classes. Each equivalence class consists of graphical models that correspond to equal test model structures, i.e. which are indistinguishable for the optimization algorithm. This feature allows us to offer the hypothesis that the optimizer operates identically on equivalent graphical models (uniformity hypothesis). Therefore, it is sufficient to have one representative of each equivalence class in a test set.

Since the set of all model structures, i.e. equivalence classes, is generally infinite, in order to obtain a representative set of tests, we should then choose a finite subset of model structures on the basis of patterns derived in the course of analysis of the optimization algorithm. Consequently, a test coverage criterion is formulated in terms of the abstract generation model.

C. *Creation of test generators*

In order to obtain a set of test cases for the optimizer under test according to the GraphOTK approach, it is necessary to create a generator. This generator will create the representative (i.e. meeting the chosen test coverage criterion) set of test cases, i.e. graphical models.

A generator consists of two components. The first component is called *iterator* and is responsible for generation of model structures one by one. The second component is called *mapper* and is responsible for mapping any model structure to a corresponding test case – the textual representation of the graphical model.

The iterator creates a set of model structures that meets the chosen test coverage criterion.

Given a model structure S, the mapper creates a corresponding test case with the following feature: the model representation of a graphical model created from S is equal to S.

After iterator and mapper are developed they are combined into a test generator which is ready to generate test cases.

IV. Case Studies

By means of two case studies, this section illustrates the application of the GraphOTK approach to optimizations which can be applied to Simulink / Stateflow models in the course of code generation.

A. *Test generator for a Switch block optimizer*

The first test suite addresses a code generator component that translates and optimizes Switch blocks which are part of the Simulink language.

A Switch block (Fig. 3) has three input signals (In1, control and In2). Each of the inputs can either be a constant value, a signal source (InPort block) or a model pattern (e.g. arithmetic expression). A Switch block defines an if-then-else control structure – it propagates either the first (In1) or the third (In2) input to its output depending on the value of the control input. If the signal on the control input is greater than or equal to a threshold parameter the block propagates the first input (In1) to its output, otherwise it propagates the third input (In2).

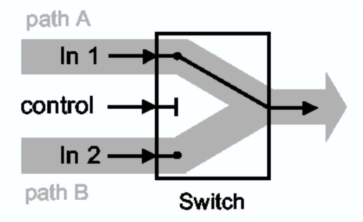


Fig. 3. Switch block

During code generation, the optimizer takes into account the range of possible values of the control signal and detects if the boolean expression «control >= threshold» (or «control > threshold», depending on block parameters) remains constant during the code execution. If so, one of the two branches of the if-then-else structure represented by the Switch block will never be executed (i.e. the same input signal of the Switch block is always transferred to output) and may be eliminated (cf. [11]).

With respect to step 1) of the GraphOTK approach, a test generation model is constructed on the basis of an abstract description of the optimization algorithm. In the case of the *Switch* block optimization, the optimization algorithm is formulated using the following terms: *Switch* block and different blocks from the Simulink library from which model patterns can be built for *Switch* block inputs. For the *Switch* block optimizer it is important to take into account those parameters of entities corresponding to the terms; namely attributes that influence the value of the output signal of blocks (for example, the amplitude of the output signal of *Sine Wave* block).

A pattern for the optimizer is a *Switch* block with different combinations of *threshold* parameter and range of possible values of the *control* signal. During the test generation Simulink models containing single subsystems with *Switch* blocks are built, and so various model patterns with different parameters are provided as input for the test.

The main parts of the resulting abstract generation model are given in Fig. 4.

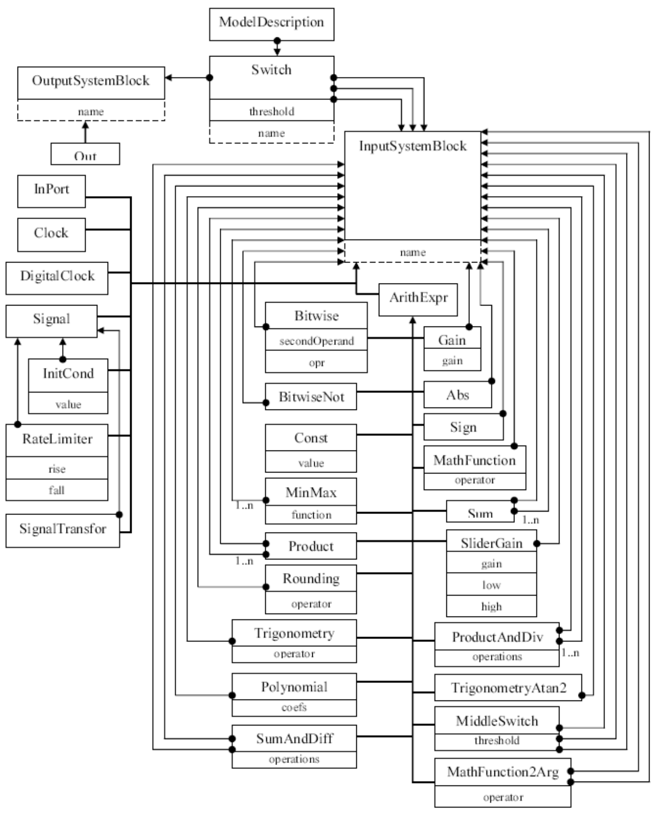


Fig. 4. Abstract generation model for Switch block optimization

According to step 2) of the GraphOTK approach, a test coverage criterion has to be defined. In the case study, the aim of the generation was to obtain tests that met the following requirements:

- Tests should contain models with Switch blocks; all inheritors of the *InputSystemBlock* node of the abstract test model should be iterated as each of the Switch block inputs;

- All inheritors of the *InputSystemBlock* node of the abstract generation model should be iterated for blocks that have one or more inputs for each input; if the input has to be a continuous signal, all inheritors of the Signal node should be iterated as this input;
- For blocks that can use different amounts of arguments with the minimal allowed amount of *N*, tests should contain models where such blocks will have *N* arguments and models where such blocks will have *N+1* arguments;
- Tests should contain models where the value of the control signal is not less than the threshold value, models where the value of the control signal is less than the threshold value and models where the value of the boolean expression «control >= threshold» does not stay constant during model execution;
- For blocks with several arguments and operations (e.g. *Sum* block where '+' or '–' can be set for each argument), tests should contain blocks with all possible operations (for example, the tests for block *Sum* mentioned above should contain blocks to which some '+' arguments will be applied, and blocks to which some '–' arguments will be applied).

A test set may be generated by generator parameters adjustment without blocks corresponding to signal sources and blocks corresponding to signal transformations.

Tests generated by default contain valid models and invalid models that may cause exceptions during execution in Simulink. The generator may be adjusted to produce only valid test models. In this case, the observance of dynamic semantics in mathematical functions is guaranteed. For this purpose, the following actions are performed during test generation:

- For mathematical functions whose range of definition is a segment, an interval or a half-interval (asin, acos, acosh, atanh, log, log10, sqrt) before the input of the Math Function block, a *Saturation* block is placed to limit the input signal according to the range of definition of the function;
- For reciprocal functions (1/x), rem functions and the *Product* block where the division is performed, the input signal is compared to zero (using the *Relational Operator* block) and the result of the comparison (1, if the value of the signal equals zero, 0 otherwise) is added to the signal;
- For *BitwiseLogicalOperator* blocks, the absolute value of the input signal is taken (using the *Abs* block) and this value is converted to uint32 type using the *DataTypeConversion* block;
- For the pow function, the check is performed if both the first argument is zero and the second argument is negative. In this case, the non-negative value is passed to the function instead of zero. (*Relational* and *Logic* blocks are used to perform the comparison and the result of the comparison is added to the first input of the Math Function block with the pow function using *Sum* block).

If invalid models are also created, the actions listed above are not performed.

The size of the test set generated in step 3) and the generation time are presented in Table I.
A commented example of a generated test with a screenshot can be found in Appendix A.

TABLE I
CHARACTERISTICS OF THE GENERATED TEST SET FOR A SWITCH BLOCK OPTIMIZER

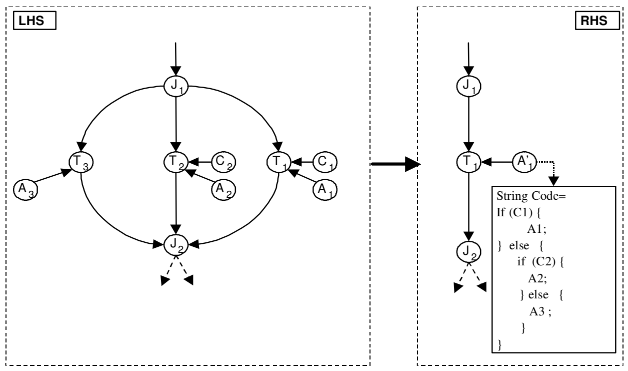
Number of tests	**Total size, in MB**	**Generation time, in s**
3 112	64	307

B. *Test generator for a Flowchart optimizer*

The second test suite is designed to test a code generator component that translates and optimizes a *Flowchart* representing an *if-then-else* control structure within Stateflow.

The flowchart optimizer searches the chart for *junctions* connected by more than one *transition node*. Each transition node must be referenced by a *condition* and can have an *action* node. One of the transitions that come out from one junction represents the "else" branch of the if-then-else control structure and is not allowed to have a condition node.

The optimizer replaces all transition nodes that connect the same junctions by one transition node. All conditions are checked and the corresponding actions executions are placed in the action node of the new transition. The example of the graph transformation performed by the optimizer is shown in Fig. 5 (cf. [11]).



```
String Code=
If (C1) {
        A1;
}  else  {
     if  (C2) {
        A2;
     } else  {
        A3 ;
     }
}
```

Fig. 5. Example of graph transformation performed by a flowchart optimizer

In the case of the flowchart optimization, the optimization algorithm is formulated using the following terms: *StateflowMachine* (a part of the Simulink Stateflow Chart block, which contains the flowchart description) and *connective junctions*.

A pattern for the optimizer is a Stateflow Chart block, where a chart contains at least two junctions connected by transition nodes.

During the test generation, different cyclic and acyclic graphs for a Stateflow Chart block are built that define if-then-else control structures of various depths. It is guaranteed that any other junction can be reached from the first junction of the graph. The number of connective junctions varies from 3 to 10. The number of transitions that connect any two junctions varies from 0 to 3.

Each transition node (except one) is referenced by a condition node that checks if the value of the input signal lies in the interval given. It is guaranteed that the conditions of the transition nodes that come out from the same junction are mutually exclusive.

Each transition node (except one) is also referenced by an action node. The action nodes form the output signal depending on the value of the input signal. On transitions that come out from the starting junction, the value of the output is set to the value of the input multiplied by some coefficient (coefficients are transition-dependent). On the other transitions, the value of the input multiplied by some coefficient is added to the output (coefficients are transition-dependent).

Each test is a Simulink model that contains a single subsystem with a Stateflow Chart block. The input signal for the Stateflow Chart block comes from the subsystem's *InPort* block, and the output signal is transferred to the *OutPort* block.

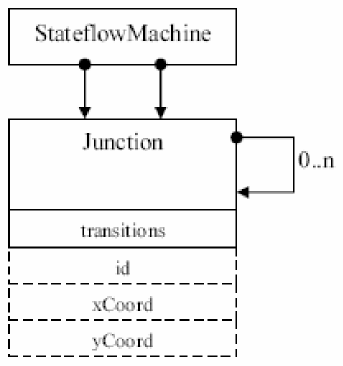


Fig. 6. Abstract generation model for a flowchart optimization

The aim of generation is to obtain test models containing the Stateflow Chart block that meet the following requirements:

- The models contain graphs with an amount of final junctions (i.e. junctions without output transitions) from 1 to 5 (the value "5" was chosen to obtain an acceptable quantity of tests);
- The models contain paths of a length of 1 to 5 (length here equals the number of transitions on the path) between the starting junction and each of the final junctions (by transitions through other junctions, not regarding the loops); all possible combinations of path lengths presented in the graph should be iterated (i.e. should be a graph where all paths have length 1; a graph where there are paths of length 1 and of length 2, and so on);
- The models contain both paths with loops and paths without loops for each path-length combination mentioned above;
- The number of transitions between the junctions on the path should vary from 1 to 3 (that corresponds to an unconditional jump, "if-else" structure and "if-elseif-else" structure).

The size of the generated test set and the generation time are presented in Table II.

A commented example of a generated test with a screenshot can be found in Appendix A.

TABLE II
CHARACTERISTICS OF THE GENERATED TEST SET FOR A FLOWCHART OPTIMIZER

Number of tests	**Total size, in MB**	**Generation time, in s**
1 335	60	225

V. CONCLUSION

This paper has suggested a test generation approach for tools that operate on various models, e.g. transforming or optimizing models. The method solves one of the main problems of testing and allows test case generation to be automated. Test generators can be adjusted to produce test sets that meet different coverage requirements (and, therefore, contain a different number of tests).

According to the approach described, test generators were created for code generators that deal with Simulink/Stateflow models. The test generators were created using the GraphOTK tool developed at ISPRAS. The generation approach was successfully applied to two exemplary transformations on Simulink and Stateflow models during code generation.

We assume that the suggested method could be applied to test generation for any tools that perform transformations of input data in order to check the correctness of these transformations if the following requirements are met:

1. There should be clear unambiguous descriptions of transformations. It is not necessary to have full descriptions; the list of structures that will be transformed is sufficient. By means of graph transformation rules, it is necessary to have the LHS part for each transformation; the RHS part is optional. However, if full description is absent, some additional information might be required to formulate coverage criteria that will guarantee a high quality of testing.
2. The input data for the system under test should have textual representation. Data itself may not be textual, but there should be a way to create an ASCII file that could be transformed into the system's input data. For example, in the case of Simulink models, it is not necessary to generate model files directly – as it is possible to generate a MATLAB script that creates a model by using Model Construction Commands.

For example, the same method as discussed in this paper has been successfully used to generate tests for optimizers of textual programming language compilers [13].

The second requirement is not critical. However, if it is not met, the development of additional mediators between abstract model structures and real data will be required.

Neither the approach suggested nor the OTK tool provide a means for test execution and results analysis. The possible methods may be different for different applications. For example, the common method of checking optimizer correctness in compiler testing is the comparison of behaviour of optimized and non-optimized programs. While the common approach used to test code generator optimizations in model-based development is to compare the behaviour of the model during simulation and the behaviour of the optimized code. For the usage of Simulink/Stateflow models, this can be easily automated by using the MTest testing tool which provides the necessary functionality for test execution and the evaluation needed in this particular test scenario.

APPENDIX A - GENERATED TEST EXAMPLES

A. *Case Study 1. Example of test generated for Switch Block optimizer*

The generated Simulink model is shown in Fig. 7.

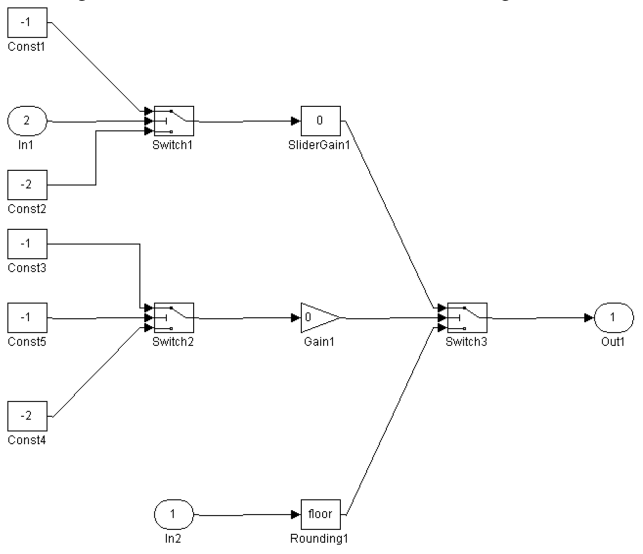


Fig. 7. Generated Simulink model for a Switch Block optimizer test (case study 1 example)

Switch2.threshold = Switch3.threshold = -1 in this model. So both blocks will transfer upper input to output and Switch2 can be removed from the model. The control signal of Switch1 cannot be analyzed with regard to only this subsystem.

B. *Case Study 2. Example of test generated for flowchart optimizer*

The flowchart of the generated model is shown in Fig. 8. This graph contains two simple cycles. Due to conditions, the machine will never loop on the upper cycle during execution, but it will always loop on the lower one (irrespective of the input signal value).

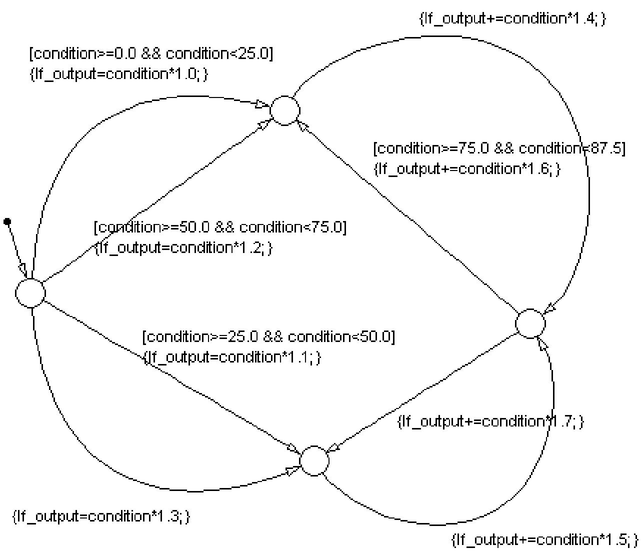

{lf_output+=condition*1.4;}
[condition>=0.0 && condition<25.0]
{lf_output=condition*1.0;}
[condition>=75.0 && condition<87.5]
{lf_output+=condition*1.6;}
[condition>=50.0 && condition<75.0]
{lf_output=condition*1.2;}
[condition>=25.0 && condition<50.0]
{lf_output=condition*1.1;}
{lf_output+=condition*1.7;}
{lf_output=condition*1.3;}
{lf_output+=condition*1.5;}

Fig. 8. Generated flowchart model for a flowchart optimizer test (case study 2 example)

REFERENCES

[1] The MathWorks. http://www.mathworks.com/
[2] ETAS ASCET. http://en.etasgroup.com/products/ascet/
[3] I-Logix. http://www.ilogix.com
[4] Potter, B.: Use of The MathWorks Tool Suite to Develop DO-178B Certified Code. ERAU / FAA Software Tools Forum. Daytona Beach, FL., May 18-19, 2004.
[5] Ranville S., Black P. Automated Testing Requirements – Automotive Perspective. // The Second Int. Workshop on Automated Program Analysis, Testing and Verification. 2001. (http://hissa.nist.gov/~black/Papers/autoTestReqsWAPATV.rtf)
[6] Conrad, M., Dörr, H., Schürr, A., Stürmer, I. Graph-Transformations for Model-based Testing. // GI-Lecture Notes in Informatics. 2002. N 12. P. 39-50.
[7] MathWorks Tools Help Land Unpiloted Boeing Spacecraft. MathWorks User Stories. (https://tagteamdbserver.mathworks.com/ttserverroot/Download/452_9797v00_Boeing_SMV_ROI.pdf)
[8] Glesner S., Geiss R., Boesler B. Verified Code Generation for Embedded Systems. // Electronic Notes in Theoretical Computer Science. 2002. 65. N 2.
[9] Grochtmann M., Grimm K. Classification-Trees For Partition Testing. // Software Testing, Verification and Reliability. 1993. N 3 (2). P. 63-82.
[10] Stürmer I. Integration of the Code Generation Approach in the Model-Based Development Process By Means Of Tool Certification. // Journal of Integrated Design and Process Science. 2004. Vol. 8 (2). P.1-11.
[11] Stürmer I. Systematic Testing of Code Generation Tools - A Test Suite-oriented Approach for Safeguarding Model-based Code Generation // PhD thesis, Technical University Berlin, 2006.
[12] Stürmer, I., Weinberg, D., Conrad, M. Overview of Existing Safeguarding Techniques for Automatically Generated Code. // 2nd Int. ICSE Workshop on Software Engineering for Automotive Systems (SEAS'05), St. Louis, USA, May 2005.
[13] Kossatchev, A.S. , Petrenko, A.K., Zelenov, S.V., Zelenova S.A. Application of Model-Based Approach for Automated Testing of Optimizing Compilers. In Proceedings of the International Workshop on Program Understanding (Novosibirsk – Altai Mountains, Russia), July 14–16, 2003, 81–88.
[14] Kuliamin, V., Petrenko, A. Applying Model Based Testing in Different Contexts.http://www.dagstuhl.de/files/Materials/04/04371/04371.KuliaminVictor.Paper.pdf.