Devising Mutant Operators for Dynamic Systems Models by Applying the HAZOP Study

Rodrigo Fraxino Araujo
José Carlos Maldonado
Márcio Eduardo Delamaro
Instituto de Ciências Mat. e de Computação
Universidade de São Paulo
São Carlos, Brazil
{*rfaraujo, jcmaldon, delamaro*}*@icmc.usp.br*

Auri Marcelo Rizzo Vincenzi
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Brazil
auri@inf.ufg.br

François Delebecque
Met., Alg. et Log. pour l'Automatique
Inst. Nat. de Recherche en Inf. et Aut.
Rocquencourt, France
francois.delebecque@inria.fr

***Abstract*—Embedded systems are increasingly present in many electronic devices. Therefore, it is necessary to use rigorous testing techniques aimed at ensuring that these systems behave as expected. Our contribution is the definition of mutant operators for the context of embedded systems models. We focus on dynamic systems models, specifically on Simulink and Scicos models, which are considered standards in many industrial application domains, such as avionics and automotive control. The HAZOP study was applied to investigate and analyze all the main features of such models, in order that the resulting mutant operators could be systematically generated. We developed a testing environment to support the mutation testing for dynamic system models, which was used to employ the defined mutant operators in a sample application.**

***Keywords*-Simulink, Scicos, HAZOP, mutation testing.**

I. Introduction

Due to the complexity of systems and the ever-increasing needs for shortening time-to-market pressures, the testing task has become even more challenging. A common problem is the testing stage being performed at the end of a project development life cycle. Thus, when faults are found, the cost to fix them is much higher [19].

A possibility to lessen the aforemetioned problem is by using precise models that support a system development life cycle. Models are concise and understandable abstractions that capture the decisions of the functions of a system whose semantics are derived from the concepts and theories of a specific domain [18].

In this context, platforms such as ScicosLab/Scicos [12] and Matlab/Simulink [20] are widely used to design and simulate dynamic system models. One of their advantages is the applications analysis at different levels of abstraction. Another benefit is the automatic code generation, which reduces development costs and programming faults. In this paper we will use *dynamic systems* aiming specifically at *Simulink* and *Scicos* systems.

To ensure the reliability of this kind of system, the industry has been investing in an approach known as model based testing [6]. In this approach, it is easier to automate the testing activity, which includes an automatic generation of test sets. The testing activity can begin to take place in a more abstract level, even before the software is coded. This leads to a more efficient process with significant cost reduction and a final product with higher quality.

In order to support this approach, our goal is to make possible the application of the mutation testing in embedded systems models, or specifically in dynamic systems models. In this paper, we show how a set of mutant operators was defined by the employment of the HAZOP (Hazard and Operability) [14] study to evaluate the features of such models. Some of these mutant operators were implemented in a testing tool that supports the mutation testing for dynamic systems models.

The mutant operators are responsible for determining the testing requirements of a model, that must be satisfied by the choice of an adequate input test set. A reason that ensures the wide usage of the mutation testing is the quality of the resulting final test set, i.e., its proneness to reveal faults [1].

In order to describe our study and the resulting mutant operators, the remainder of this paper is structured as follows. Section II describes dynamic systems models and the HAZOP study. In Section III, we show how the HAZOP study was employed in dynamic systems models. Section IV presents the mutant operators generated by a rigorous analysis of the achieved results. In Section V, a testing tool to support the mutation testing is described along with a sample application regarding the employment of our defined mutant operators. Section VI presents a discussion regarding related work. Section VII concludes with some final remarks and an outlook on future directions.

II. Background

Mutation Testing is a testing approach in which the product under test is altered several times, creating a set of alternative products with slight syntactical differences, the so-called mutants. The tester is responsible for choosing test data that show difference in the behavior among the original product and the mutant products [16]. The test set quality is measured according to its likelihood of revealing faults [9].

The construction of mutant operators must be driven by an analysis of the characteristics of the product under test. A great deal of authors do not employ general guidelines and a rigorous methodoloy for their definition. The mutant operators are usually a representation of a fault model considering the underlying product [10]. In our case, we are exploring a larger number of mutant operators generated by systematically applying the HAZOP study in dynamic systems models, which may later be minimized by the conduction of experiments.

In the following subsections we present a brief overview of a dynamic system model and of the HAZOP study. We used it to analyze the features of a dynamic system model, making possible to define appropriate mutant operators, that can guide the test data generation process for this sort of model.

A. Dynamic System Model

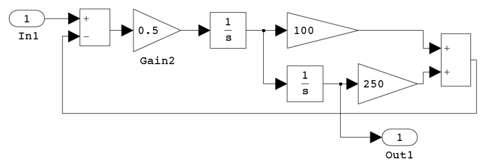
A dynamic system consists of a set of possible states, together with a rule that determines the present state from a past state. According to Korn [15], dynamic systems relate model-system states to earlier states. Classical physics, for example, predicts continuous changes of quantities such as position, velocity, or voltage with continuous time.

With the increasing complexity of these systems, development tools have become imperative to support their design. Simulink [20] and Scicos [12] are environments for sharing data, designs and specifications, making possible to develop more reliable critical systems and safely generating code. They are widely used within industry due to the large expressiveness of their languages.

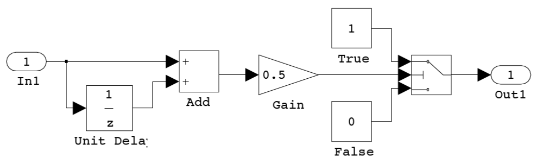
The models used by such environments are based on block diagrams. These blocks include a library of sinks, sources, connectors and linear and non-linear components. Models can be hierarchical, which helps to understand the model organization and how the components interacts with each other [20, 12].

Such platforms offer a convenient way to describe systems that evolve according to time. Such systems are mathematically represented by systems of equations, that are differential equations in the case of continuous time systems, difference equations in the case of discrete time systems, and a mix of both in the case of hybrid systems. The simulation of these types of systems is based on numerical algorithms, where the solution of a system of equations, i.e., the semantics of a dynamic system model, is given by the sequence of values representing the temporal functions [7]. The input values can be read from a file or provided by a signal generator, e.g., a sinusoid or a square wave generator.

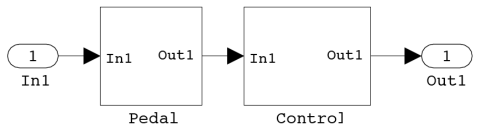
Figure 1 contains an example of a dynamic system model that is divided into three subsystems [7]. A continuous time subsystem is present in Figure 1a and represents a braking pedal as a mass-spring-damper mechanical system. A discrete time subsystem is present in Figure 1b and is responsible for detecting when the pressing force is greater than a given threshold to activate the brake. Figure 1c presents the main system, a composition of both subsystems, containing an input, the force, and an output, the detection result.



(a) Continuous Time Model



(b) Discrete Time Model



(c) Hybrid Model

Figure 1: Dynamic Systems Models

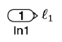
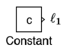
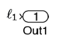
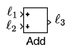
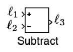
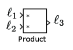
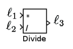
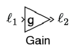
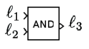
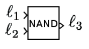
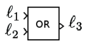
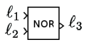
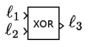
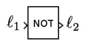
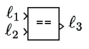
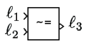
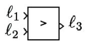
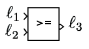
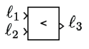
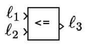
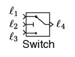
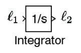
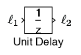
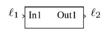
These models are composed by blocks connected by lines (signals). The blocks can be elementary, containing simple operations (as arithmetics, for instance), or subsystems, that contains a composition of elementary blocks. In the models of Figure 1, it is worth emphasizing the *Integrator* and the *UnitDelay* blocks, which introduce the notion of time. When an Integrator is used, the model is called of continuous time, and the operation associated to the block is a mathematical integration over time. A model that uses a UnitDelay is called of discrete time. A mix of both produces a hybrid model, defined as a data flow where the signals are continuous or discrete time functions.

A block worth mentioning is the *Switch*, which contains two data inputs and one control input. The developer must specify how the evaluation of the second input (control) must be performed, in order to redirect the first or third data input to the output. Thereby, this block can be compared to an if-then-else sentence. Table I, adapted from Chapoutot and Martel [7], presents the main blocks of a dynamic system model. VIADINHO

B. HAZOP (Hazard and Operability Study)

Hazard and operability studies (HAZOP) [14] originated in the chemical industry and, thereafter, have been widely

Table I: DYNAMIC SYSTEM BLOCKS

Name	Block	Descript.	Equation
Input		Input	l_1 = In(t), $\emptyset$
Const.		Constant	l_1 = c, $\emptyset$
Output		Output	Out(t) = l_1, $\emptyset$
Add		Adittion	l_3 = l_1 + l_2, $\emptyset$
Sub		Subtraction	l_3 = l_1 - l_2, $\emptyset$
Product		Multipl.	l_3 = l_1 * l_2, $\emptyset$
Divide		Division	l_3 = l_1 / l_2, $\emptyset$
Gain		Multipl. by Constant	l_2 = g * l_1, $\emptyset$
AND		AND	l_3 = l_1 && l_2, $\emptyset$
NAND		NAND	l_3 = !(l_1 && l_2) ,$\emptyset$
OR		OR	l_3 = l_1 || l_2, $\emptyset$
NOR		NOR	l_3 = !(l_1 || l_2), $\emptyset$
XOR		XOR	l_3 = (l_1 && !l_2) || (!l_1 && l_2), $\emptyset$
NOT		NOT	l_2 = !(l_1), $\emptyset$
==		==	l_3 = (l_1 == l_2), $\emptyset$
~=		! =	l_3 = (l_1 != l_2), $\emptyset$
>		>	l_3 = (l_1 > l_2), $\emptyset$
>=		>=	l_3 = (l_1 >= l_2), $\emptyset$
<		<	l_3 = (l_1 < l_2), $\emptyset$
<=		<=	l_3 = (l_1 <= l_2), $\emptyset$
Switch		Conditional Command	l_4 = if(ρ(l_2), l_1, l_3), $\emptyset$
Integr.		Continuos Time Integration	$l_2(t)$ = $\eta(t)$, $\dot{\eta}(t)$ = $l_1(t)$
Unit Delay		Discrete Time Delay	$l_2(t)$ = $\eta(t)$, $\eta(t+1)$ = $l_1(t)$
Sub System		Subsystem	l_2 = f(l_1), $\emptyset$

applied in different contexts to assess varying sorts of systems. The main purpose of such studies is to systematically examine the behavior of the underlying system in order to determine deviations and hazards that might arise as well as potential related problems. They are currently used in several areas for qualitative risk analysis [3].

The first step in the HAZOP study consists in identifying entities and attributes of the system under examination by means of an analysis of its description. For instance, taking a software system into consideration, such a description can be the software control flow. The next step is to apply a number of predetermined *guidewords* to system attributes in order to investigate possible deviations and determine possible causes and consequences [13].

The role of these guidewords is to act as mnemonics. After structurally applying each of them to attributes of the system under examination, it is possible to focus on a certain sort of anomalous behavior and ponder over it. Thus, this method provides additional insight into potential deviations. However, matching a guideword with an attribute requires interpretation. Depending on the context, guidewords may have more than one interpretation. For instance, *MORE* applied to a data value attribute can be interpreted as *greater*, i.e., yielding a greater value then it should be. Similarly, applying *MORE* to bit rate attributes can be interpreted as *higher*. Moreover, guidewords may be meaningless in certain contexts, demanding the creation of additional guidewords.

III. HAZOP IN A DYNAMIC SYSTEM MODEL

The testing activity is typically applied taking in consideration source code, platform independent intermediate representations or machine-specific code. However, several researches propose its use in a representation at a higher level of abstraction, i.e., models [16]. In our case, we address the testing of Simulink and Scicos models by applying the HAZOP study to the specification of a dynamic system model.

The representation examined is the syntax of the model construction. Attributes are identified for each construct of a dynamic system model, and syntactic deviations are investigated by the employment of *guidewords* to these attributes. For each possible deviation, the cause and consequence of a deviation are examined in order that mutant operators, that result in minor syntactic modifications, can be derived [13].

Table II presents the identified attributes for a dynamic system model. To show how the employment of the HAZOP guidewords to the attributes of a dynamic system model were performed, we present some examples as follows.

In the first example the construct *types* and the atribute *compatibility* affect the blocks *Input* and *Output*. It is possible to apply 2 guidewords:

- AS_WELL_AS. Cause: replacement among compatible types among double, single, int8, uint8, int16, uint16,

Table II: ATTRIBUTES OF A DYNAMIC SYSTEM MODEL

Constructs	Attributes	Related Blocks
Types	Compatibility Intervals	Input Output
Variables	Stored Values	Lines
Constants	Stored Values	Constant
Blocks	Execution Result of Switch Statement	Switch
	Execution Result of Temporal Statement	UnitDelay Integrator
	Interaction Among Subsystems	Subsystem
Expressions	Evaluation Result of Relat. Op.	Relat. Op.
	Evaluation Result of Logic. Op.	Logic. Op.
	Evaluation Result of Arith. Op.	Arith. Op.

int32, uint32 and boolean. Consequence: no loss of information.

- PART_OF. Cause: types with lower capacity can be used as, for instance, single instead of double. Consequence: it is possible to lose information or precision.

In the next example the construct *constants* and the atribute *stored values* affect the block *Constant*. It is possible to apply 3 guidewords:

- MORE. Cause: increase of a numeric value. Consequence: possible incorrect result.
- LESS. Cause: decrease of a numeric value. Consequence: possible incorrect result.
- OTHER_THAN. Cause: replacement among the constants of a model. Consequence: possible incorrect result.

Similar to the aforementioned examples, the *guidewords* were applied to the defined attributes of a dynamic system model, resulting in the analysis of all main blocks for this kind of model. Due to the lack of space, we are not able to present the relations among all guidewords and the defined mutant operators.

IV. MUTANT OPERATORS DEFINITION

A set of mutant operators was derived from the employment of the HAZOP study in a dynamic system model and is presented in this section. It is important to note that not all *guidewords* resulted in a mutant operator, because according to our evaluation, in some occasions the operation would not be significant, or would always result in a faulty model impossible to be simulated.

We decided to keep a conservative approach in the definition of mutant operators, i.e., all coherent mutant operators possible to be derived for this kind of system by the application of the HAZOP study were defined.

Types

Type Replacement Operator
This operator replaces a type with compatible types, and can be applied directly in the Input and Output blocks, which are used in the interaction among systems and subsystems.

Variables

Variable Change Operator
This operator acts in the connections among the blocks of a model, increasing or decreasing the value that is being carried. As it is not possible to know *a priori* which value that is, a possible implementation is to insert an *add* or *subtract* block between the source and destination blocks.

Variable Replacement Operator
This operator acts in the connections among the blocks of a model, replacing the compatible values that are being carried by swapping their connections. For the implementation, special attention must be drawn to the compatibility analysis among the number of inputs and outputs of each block.

Constants

Constant Change Operator
This operator is responsible for increasing or decreasing the value of the constants of a model.

Constant Replacement Operator
This operator replaces the values among the constants of a model.

Blocks

Statement Swap Operator
This operator is responsible for swapping the first and the third input of the Switch block, acting in a way similar to modifying the evaluation result of the blocks condition.

Delay Change Operator
This operator can increase or decrease the delay in which the output of the UnitDelay or the Integrator blocks will be provided to the system.

Subsystem Change Operator
This operator swaps the connections between two subsystems or between a main system and a subsystem aiming to act in the integration of components of a model. Despite being a suboperator of the Variable Replacement Operator (VRO), this operator may be useful if the tester desires to analyze only the interaction among the subsystems of a model.

Block Removal Operator
This operator is responsible for removing each of the blocks of a model, and can be useful to ensure that every block is being used and that a test data exists to force its execution.

Expressions

Relational Op. Replacement Operator
This operator is responsible for the replacement among the relational operators $>$, $>=$, $==$, $\sim=$, $<$ e $<=$.

Arithmetic Op. Replacement Operator
This operator is responsible for the replacement among the blocks Add, Sub, Product, Divide and Gain.

Logical Op. Replacement Operator
This operator is responsible for the replacement among the logical operators AND, OR, NAND, NOR, NOT and XOR.

A. Summary

12 mutant operators were defined by employing the HAZOP study for Simulink-like models and are summarized in Table III. Most of the defined mutant operators deal with the data flow of a Simulink-like model, which is the essence of this type of system. Three mutant operators were defined aiming at dealing with unique features of this kind of model. Although most of the operators deals with modification in a model, in certain cases for their implementations new blocks need to be added and removed. As a result, we consider that they are a complete set, taking into account that along with our analysis, the defined mutant operators force addition, alteration and deleting operations in Simulink-like models.

Table III: MUTANT OPERATORS

Acron.	Description
TRO	Type Replacement Operator
VCO	Variable Change Operator
VRO	Variable Replacement Operator
CCO	Constant Change Operator
CRO	Constant Replacement Operator
SSO	Statement Swap Operator
DCO	Delay Change Operator
SCO	Subsystem Change Operator
BRO	Block Removal Operator
RORO	Relational Op. Replacement Op.
AORO	Arithmetic Op. Replacement Op.
LROO	Logical Op. Replacement Op.

The first one, SSO, aims to swap the inputs of a Switch block, altering the control flow of a system. The second operator, DCO, deals with the temporal characteristics of a system, and acts in the UnitDelay and Integrator blocks. The third one, SCO, operates in the interaction among the subsystems of a model, swapping the blocks connections among them or among a main system and possible subsystems.

Table IV presents the worst case scenario, or the maximum number of mutants to be generated by each mutant operator regarding the model property that is being affected, i.e., input and output ports, blocks or blocks connections (lines). The VRO mutant operator is the most likely to produce a larger number of mutants.

Table IV: NUMBER OF GENERATED MUTANTS

M. Op.	Worst Case Scenario
TRO	(Inputs + Outputs) * Data Types
VCO	Lines * 2
VRO	Lines * (Lines -1)/2
CCO	Constants * 2
CRO	Constants * (Constants-1)/2
SSO	Switchs * 2
DCO	Delays * 3
SCO	SSLines * (SSLines -1)/2
BRO	Blocks
RORO	Relat. Op. * 5
AORO	Op. * $2^{Op.Inputs}$ + Gain * 2
LROO	Logic. Op. * 5

V. TESTING TOOL

TeTooDS (Testing Tool for Dynamic Systems) [2] can interpret dynamic systems models, interact with simulation environments such as Scicos or Simulink, and is used to assist in the test data generation task. It was previously developed to provide support for the application of functional criteria, specifically the pairwise approach, in dynamic systems models. This approach ensures that any two possible values, belonging to two different parameters, will be present in at least one test data [11].

We have extended TeTooDS to support mutation testing in dynamic systems models. The first necessary step was the development of a full-blown parser, that provides the information required by the mutant operators to the generation of mutants of a model. These information include input ports, input datatypes, blocks, blocks parameters, connections and output ports. Our parser makes use of the pyparsing module [17], a flexible approach for creating and executing grammars, against the lex/yacc approach or the use of regular expressions. The pyparsing module provides a library of classes that supports building grammars directly into the Python code.

After parsing a Scicos or Simulink model, which is accomplished when a testing project is created in TeTooDS, several options become available to the tester. A possibility is to select which mutant operator will be used for the generation of the mutant models.

The tester can also visualize the mutant models inside TeTooDS: *(i)* as an image; *(ii)* as the source code of the model; or *(iii)* using TeTooDS to call Scicos/Simulink along with the mutant model. It is useful for performing an analysis of equivalent mutant models or to see which mutants are alive or dead.

Test cases can be added by specifying input files that will be read by the dynamic system model during its simulation, together with the specification of which output files should be read by the testing tool when the simulation finishes.

To run the simulation of the main dynamic system model and the generated mutants, TeTooDS provides a default script that can be used or customized in order that the parameters values, such as start time, stop time and step time, can be configured according to the tester needs. After the simulation finishes, output files are analyzed and the mutation score is updated with the mutants status information.

A. Sample Application

This model represents an electronic regulator which contains a flow regulator, a temperature sensor and a logic controller. The system has three input ports: temperature, temperature lower bound and temperature upper bound. When the temperature is below the lower bound, a valve is closed, i.e., receives a zero value. When the temperature is above the high bound, a valve is opened, receiving a value of 100. When the temperature is between these limits, the valve aperture is calculated by the expression $(5.0/3.0) * (temperature - low\_bound)$ [4].

We used all the defined mutant operators, applying one mutation at a time, which resulted in 131 generated mutants. For the execution of the mutants, firstly we selected input data randomly. For the remaining mutants, in order to achieve 100% of mutation score, we manually analyzed each mutant aiming to select a test data that could kill it or mark it as equivalent. Table V shows the number of mutants generated by each operator.

Table V: NUMBER OF MUTANTS

Operator	Mutants	Operator	Mutants
TRO	0	DCO	4
VCO	36	SCO	0
VRO	26	BRO	19
CCO	4	RORO	10
CRO	1	AORO	15
SSO	1	LROO	15

The TRO did not return any mutants, as we used a Scicos model as source and it does not allow the use of several data types. Mutants also were not generated by the SCO operator, as the number of inputs of the subsystems of this particular model are not compatible.

To show the viability of the defined mutant operators, the second step of our case study was to manually generate the C code that corresponds to this particular model. We used Proteum [8] and its 73 mutant operators to generate mutants for the C code, which resulted in 1473 mutants. By applying the test set that was selected to achieve 100% of mutation score in the model, which represents a simulation of the system that is going to be hardware integrated, we could achieve 98.1% in the C code.

Our first intention was to use the code that can be automatically generated by Scicos. Nevertheless, it presents too many unused variables and other pieces of unexecuted code, resulting in a large number of equivalent mutants to be analyzed (up to 100 000 mutants).

We consider that we achieved a high mutation score for this particular model when applying the test set responsible for achieving a full coverage of the model (100%) in the C generated code (98.1%), which encourages the development of a thorough experiment, taking into account all the necessary validity levels. We emphasize that our intention is to assess the feasibility of all mutation operators aiming at possible refinements.

VI. Related Work

The existing literature shows that the mutation criterion is very effective for revealing faults of traditional programs and models. Nonetheless, this criterion has not been widely explored for the context of dynamic systems models.

We are aware of two studies that aim at applying the mutation testing in dynamic systems models. The first one is described by Brillout et al. [5]. They developed a methodology to assess the correctness of Simulink models by automating the test data generation activity. Their objective is to cover the requirements imposed by the mutation testing. In order to generate and optimize the test data, the approach focus on model checking techniques. However, the authors do not clearly present an solution of how to apply the mutation testing, i.e., which mutant operators should be used to generate the testing requirements.

The second study is the one of Zhan and Clark [21]. Despite introducing a testing framework for Simulink models and focusing on the mutation testing, the approach presents a few limitations. The authors make use of a random test data generator and try to improve the test set by the use of dynamic analysis and simulated annealing methods, in order to satisfy the constraints imposed by their mutant operators. We consider as a drawback of their approach the low number of defined mutant operators, i.e., *add, multiply* and *assign*. In our approach, we have tried to overcome such issue by performing a systematic analysis of a dynamic system model in order to define a complete set of mutant operators for this context, that includes the ones defined by Zhan and Clark.

VII. Final Remarks and Future Work

We address the testing of Simulink and Scicos models. Dealing with these models entails properly concerning their domain specific language, which is geared towards code generation, and also present specific features, as temporal and combinatorial characteristics.

The employment of the HAZOP study to derive mutant operators for a particular type of system can produce different syntactic variations, which can assist in finding possible faults of a system. In this paper we presented the solutions that the authors consider appropriate for dynamic systems models.

One of the advantages of the HAZOP study is that the set of mutant operators can be more complete than those generated based only on the experience of faults of a developer, since the language constructs are analyzed.

Future work also includes the definition of a method for the automatic generation of test data for dynamic systems models, that aims at satisfying the mutation test requirements. Longer term future work includes the conclusion of an integrated testing environment that can assist in the automation of the testing activity for dynamic systems models.

VIII. Acknowledgment

The authors would like to thank the financial support provided by CNPq (grant number 141976/2008-0). We are also thankful to Vinicius Durelli, who proofread and commented on drafts of this paper.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th ICSE*, pages 402–411. ACM Press, 2005.

[2] R. F. Araujo and M. E. Delamaro. TeTooDS - Testing Tool for Dynamic Systems. In *Tools Session – Brazilian Software Engineering Symposium*, Brazil, 2008. SBC.

[3] J. S. Arendt and D. K. Lorenzo. *Evaluating Process Safety in the Chemical Industry. A user guide to quantitative risk analysis*. AIChE, second edition, 2000.

[4] M. Blackburn, R. Busser, and A. Nauman. Why model-based test automation is different and what you should know to get started. In *International Conference of Practical Soft. Quality and Testing*. SPC, 2004.

[5] A. Brillout, M. He, Nannan afend Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher. Mutation-based test case generation for simulink models. In *Proceedings of the 8th international conference on Formal methods for components and objects*, FMCO'09, pages 208–227, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[7] A. Chapoutot and M. Martel. Abstract simulation: A static analysis of simulink models. In *ICESS '09: Proceedings of the 2009 International Conference on Embedded Software and Systems*, pages 83–92, Washington, DC, USA, 2009. IEEE Computer Society.

[8] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for c programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, 1996.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[10] F. Ferrari, J. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *Software Testing, Verification, and Validation*, pages 52 –61, april 2008.

[11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies - a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[12] INRIA Rocquencourt. Scicos, 2011. Available at http://www.scicos.org.

[13] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of java mutation operators using hazop. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, 1999.

[14] T. Kletz. *Hazop and Hazan: Identifying and Assessing Process Industry Hazards*. CRC Press, fourth edition, 1999.

[15] G. A. Korn. *Advanced Dynamic-system Simulation: Model-replication Techniques and Monte Carlo Simulation*. Wiley-Interscience, 2007.

[16] A. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[17] P. McGuire. Pyparsing, 2011m. Available at http://pyparsing.wikispaces.com.

[18] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating Simulink models into input language of a model checker. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.

[19] W. Perry. *Effective methods for software testing, third edition*. John Wiley & Sons, Inc., 2006.

[20] The Mathworks Inc. MATLAB and Simulink, 2011. Available at http://www.mathworks.com.

[21] Y. Zhan and J. A. Clark. A search-based framework for automatic testing of matlab/simulink models. *Journal of Systems and Software*, 81(2):262–285, 2008.