Stochastic Error Propagation Analysis of Model-driven Space Robotic Software Implemented in Simulink

Andrey Morozov, Klaus Janschek
Technische Universität Dresden
Nöthnitzer Str. 43,
01187 Dresden, Germany
{andrey.morozov,
klaus.janschek}@tu-dresden.de

Thomas Krüger, André Schiele
European Space Agency
ESTEC, Keplerlaan 1,
2201AG Noordwijk, The Netherlands
{thomas.krueger,
andre.schiele}@esa.int

ABSTRACT

Model-driven software development methods are widely used in safety-critical domains including space robotics. The MATLAB Simulink environment is the common choice of control engineers. This article introduces a new method for a fully automatic transformation of a Simulink model to a dual-graph model for stochastic error propagation analysis. The error propagation analysis provides important inputs for system reliability methods, required by industrial standards such as FTA and FMEA. The dual-graph error propagation model is a mathematical abstraction of key system design aspects that influence error propagation processes: control flow, data flow, and component-level reliability properties. This model helps to estimate the likelihood of error propagation to hazardous system parts and quantify the negative impact of a fault in a particular component on the overall system reliability. In praxis, the manual creation of an error propagation model of a complex system requires a huge effort. The transformation method, introduced in this article, is a fast and promising solution. The method is demonstrated as a part of a stochastic analysis of a real-world model-driven space robotic software.

CCS Concepts

•Computing methodologies → Model development and analysis; •Computer systems organization → Reliability; •Hardware → Fault tolerance; *Transient errors and upsets; Safety critical systems;* **•Mathematics of computing** → Markov processes; **•Software and its engineering** → Model-driven software engineering;

Keywords

Error propagation model, space robotic software, model-driven software, Simulink, model transformation, model-based analysis, control flow, data flow.

MORSE '16, July 01 2016, Leipzig, Germany

DOI: http://dx.doi.org/10.1145/3022099.3022103

1. INTRODUCTION

Model-driven software development methods are widely used in the safety-critical domains including space robotics. MATLAB Simulink is a common development environment and a preference of control engineers. Simulink is based on block diagrams that follow principles of a classical mathematical model of control engineering - State Space Representation. A block diagram represents a control algorithm as a system of functional blocks and emphasizes the data (signal) flow between them. Simulink provides a rapid prototyping tool-chain that allows automated generation of executable code, and its deployment on a target hardware system. This model-driven approach to control software development ensures consistency between the model and production code and helps to avoid many faults that could be introduced in case of conventional, manual, software development.

However, even certified [11, 10] and well-tested model-driven software is vulnerable to hardware faults. Single event upsets or electromagnetic interference [24, 19, 17] may cause silent data corruption, especially in space applications. Such faults result in erroneous outputs of the host blocks. Occurred data errors can propagate further both between blocks (from an output of one block to an input of another block) and through blocks (from an erroneous input to outputs of the same block). The analysis of this complex process is helpful in a wide range of analytical tasks associated with dependable systems development. Particularly, it will provide numerical inputs for system reliability methods, required by industrial standards e.g. failure modes and effect analyses (FMEA), hazard and operability studies (HAZOP), fault trees analysis (FTA), event trees (ET) etc.

The recently introduced [12, 13, 14, 9] dual-graph error propagation model (DEPM) is a mathematical abstraction of key system design aspects that influence error propagation processes: control flow, data flow, and component-level reliability properties. The DEPM helps to estimate the likelihood of error propagation to hazardous system parts and quantify the negative impact of a fault in a particular component on the overall system reliability.

However, the manual creation of an error propagation model of a complex system requires a huge effort. A new method for fully automatic transformation of a Simulink model to a DEPM, introduced in this article, is a fast and promising solution.

The article is organized as follows. Section 2 discusses relevant existing transformation methods for Simulink models. Section 3 gives an overview of a space robotic system and a

particular Simulink model that is used as a reference example in this article. Section 4 describes the target dual-graph error propagation model. Section 5 introduces the transformation method itself, discusses several implementation highlights, and gives an example of the method application using the reference Simulink model.

2. RELATED WORK

A number of available transformation methods from MATLAB Simulink models to other models and abstractions are discussed in this section.

A transformation method from Simulink to Lustre is presented in [2, 22, 3]. The transformation comprises three parts: type interface, clock inference, and hierarchical bottom-up translation. Lustre [8] is a declarative, synchronous dataflow programming language. SCADE (Safety Critical Application Development Environment) is a tool suite, based on the Lustre, for critical control software, extended with powerful model-checkers and test generators.

UML [16] is the software engineering based modeling paradigm with its strengths in the semi-formal specification of complex systems at a rather abstract level. In contrast, the control engineering based Simulink paradigm defines functional structures and their inherent semantics at a very detailed and concrete level. This makes these paradigms rather complementary. Structural and behavioral mapping of a Simulink model to several UML diagrams is introduced in [21]. The structure is mapped into a UML Composite Structure Diagram. The behavioral aspects, taking into the Simulink engine itself, are mapped into UML Activity Diagrams. Another Simulink to UML transformation method is presented in [18]. Blocks and lines are extracted from the Simulink model and translated into UML Class Diagrams. Behavioral aspects are not considered.

A transformation method form Simulink to Functional Blocks [23] is presented in [7]. The method is based on block-to-block structural mapping and analysis of Simulink execution priorities. Well defined, formal transformation approach of Simulink and Stateflow models into hybrid automata is introduced in [1]. The transformation is specified and implemented using a metamodel-based graph transformation tool. The transformation from Simulink to BIP (Behavior, Interaction, Priority) formalism, for modeling, analysis and implementation of heterogeneous real-time systems, is presented in [20]. Finally, our recent approach for automatic transformation of a UML Activity Diagram to a DEPM model is presented in [4].

The transformation method, introduced in this article is also focused on the analysis of structural and behavioral properties of Simulink models and thus has similarities with all the methods discussed above. The main difference, besides the specific target model, is that the introduced method involves not only static parsing of a Simulink model but also its instrumentation and simulation, which is required in particular for estimation of the component-level reliability properties. In contrast to the discussed methods, which are realized as stand-alone Simulink model parsers, our method is implemented as a collection of MATLAB scripts that heavily exploits the Simulink API. From our point of view, direct parsing of Simulink model files, is rather pointless, because of the complex, poor documented, and frequently changing format. Moreover, this functionality is already available in the Simulink API.

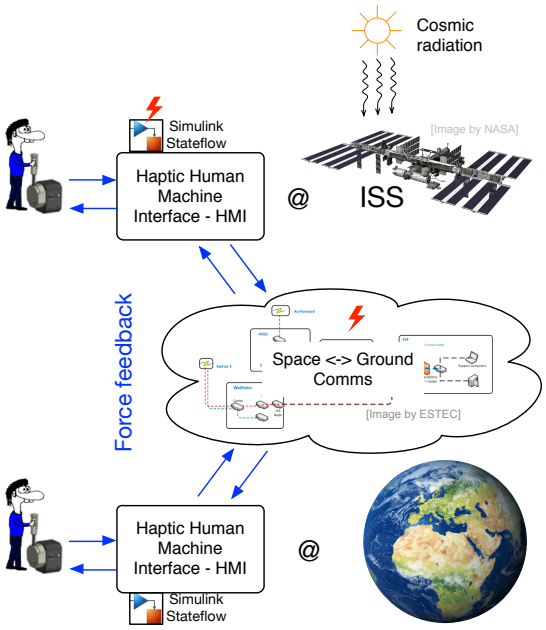

Figure 1: A simplified description of the HAPTICS-2 experiment.

3. REFERENCE SYSTEM

METERON (Multi-Purpose End-To-End Robotic Operation Network) is a suite of experiments conducted by the European Space Agency (ESA) in order to validate advanced telerobotics technologies for the usage in space [5]. The main goal is to teleoperate a robot on a planetary surface from an orbiting spacecraft. HAPTICS-2 [6] is one of the METERON experiments; its architecture is illustrated in Figure 1. The endpoints are two identical haptic devices, each consisting of a one degree of freedom (1-DoF) force feedback joystick equipped with a tablet computer: One is on board of the International Space Station (ISS) operated by an Astronaut and the other on the ground. The communication between the devices is via the Tracking and Data Relay Satellite System (TDRSS). The goal of the HAPTICS-2 is to analyze various aspects of force feedback control in space, including perception in zero-g, time delay/loss of signal and practical implications.

The joystick controllers were implemented with MATLAB Simulink models that have been accompanied by the test input data and provided for the DEPM-based stochastic error propagation analysis. Figure 2 shows a part of the Simulink model that performs cogging compensation. The original cogging compensation model has been extended with an additional *MotorVelIntegrator* block in order to demonstrate more capabilities of the introduced transformation method. This model will be used as a reference system in this article.

The model has three inputs (*In_MotorPos*, *In_MotorVel*, and *In_TorqueCmd*) and two outputs (*Out_PosError* and *Out_CurrentCmd*). The analytical task is to estimate the number of errors in the outputs (highlighted in yellow) given the probabilities of errors in the inputs. The error-prone inputs are highlighted in red and annotated with the probabilities of error occurrence.

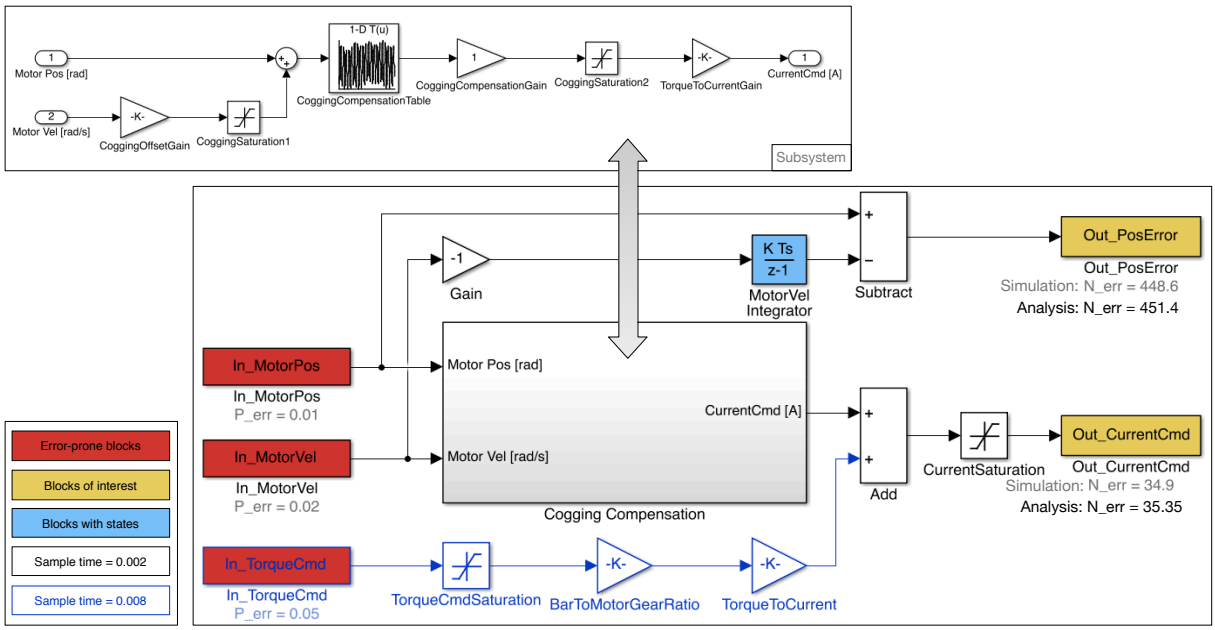

Figure 2: The reference Simulink cogging compensation model. The model is annotated with the probabilities of error occurrence (*P_err* near the red blocks). The mean number of errors will be computed for the blocks highlighted in yellow.

4. DUAL-GRAPH ERROR PROPAGATION MODEL

This section gives the set-based mathematical notation of the target dual-graph error propagation model:

$$DEPM := \langle E, D, A_{CF}, A_{DF}, C \rangle$$

E	is a non-empty set of elements;
D	is a set of data storages;
A_{CF}	is a set of directed control flow arcs, extended with control flow decision probabilities;
A_{DF}	is a set of directed data flow arcs;
C	is a set of conditions of the elements.

A simple DEPM is shown in Figure 3 a). Elements A, B, and C represent executable parts of the system. Each element has a zero or more data inputs and outputs. Data storages *d1*, *d2*, *d3*, and *output* represent variables, which can be read or written by the elements. During execution of an element, e.g. element A that is highlighted in red, errors can occur and propagate to its data outputs. The incoming errors can propagate from the inputs to the outputs depending on the internal properties of the element, defined with the probabilistic conditions, see Figure 3 b).

The control flow arcs (black lines) connect the elements. Each control flow arc is weighted with a transitions probability: After the execution of A, B will be executed with probability 0.7 and C with probability 0.3.

The data flow arcs (purple lines) describe data transfer between the elements. A data flow arc connects an element with a data storage or vice versa. The data flow arcs are considered to be the paths of data error propagation.

ErrorPro [15] is a software tool for stochastic error propagation analysis, based on the described DEPM. This tool automatically creates discrete time Markov chain models using the DEPM representation and computes required numerical reliability properties. For instance, the mean number of errors in the data storage *output* during 100 steps (execution of one element is one step) is equal to 3.63 as it is shown in Figure 3 c). A DEPM model can be stored in an XML file. A fragment of an XML file of the DEPM from Figure 3 is shown in Listing 1.

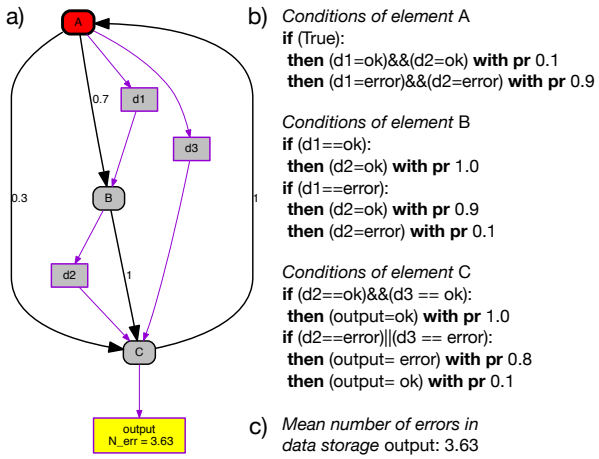

Figure 3: An example of a simple dual-graph error propagation model: a) a graphical representation b) probabilistic conditions of the elements that describe their reliability properties c) the analytical result - the mean number of errors in the data storage *output*.

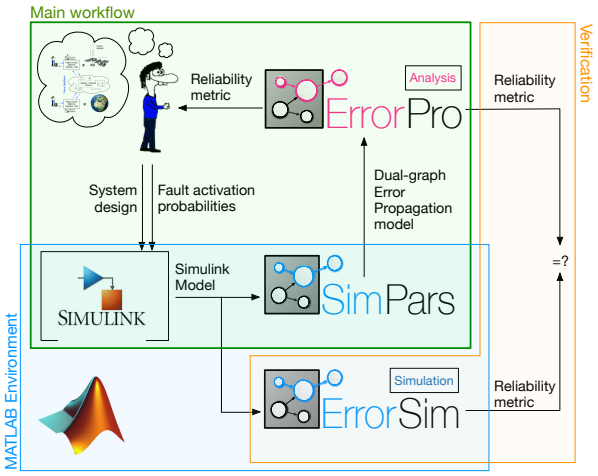

Figure 4: An analytical workflow that involves the introduced transformation method. The intended workflow is highlighted with green. SimPars is a software implementation of the transformation method. ErrorSim is a Simulink-based error propagation simulator that is used for verification.

```
<?xml version="1.0" encoding="utf-8"?>
<model n_steps="100" name="ABC Model" version="3.0">
 <element initial="true" name="A"/>
 ...
 <data name="d1"/>
 ...
 <control_flow from="A" prob="0.7" to="B"/>
 ...
 <data_flow from="A" to="d1"/>
 ...
 <conditions element_name="A">
  <if statement="True">
   <then prob="0.9" update="d1 = ok; d3 = ok;"/>
   <then prob="0.1" update="d1 = error; d3 = error;"/>
  </if>
 </conditions>
 ...
</model>
```

Listing 1: An example of a DEPM XML file.

5. TRANSFORMATION METHOD

Figure 4 demonstrates an analytical workflow that involves the proposed transformation method. The prototypical implementation of the transformation method is called SimPars. The indented workflow is highlighted with the green background: A developer creates a Simulink model, specifies the probabilities of errors for some blocks, transforms the Simulink model to a DEPM XML file using SimPars, and finally analyzes the DEPM model with ErrorPro.

SimPars and ErrorSim are implemented as MATLAB scripts and should be run in the MATLAB environment (highlighted in blue). ErrorPro is a standalone software tool. ErrorSim is a Simulink-based error propagation simulator that is used for verification of the entire analytical method. The verification workflow is highlighted in yellow.

This article is focused on the transformation method that consists of three main steps: (i) control flow analysis, (ii) data flow analysis, and (ii) block-level analysis.

5.1 Control flow analysis

The control flow analysis is the first step of the transformation method. The set of DEPM elements E and the set of control flow arcs A_{CF} are created during this step.

Simulink models consist of blocks. A block can be either non-virtual or virtual. Non-virtual blocks play an active role and define the functional behavior of the system e.g. Gain, Product, Integrator etc. Virtual blocks, by contrast, play no active role, but organize a model graphically e.g. Mux, Bus Creator, Subsystem, etc.

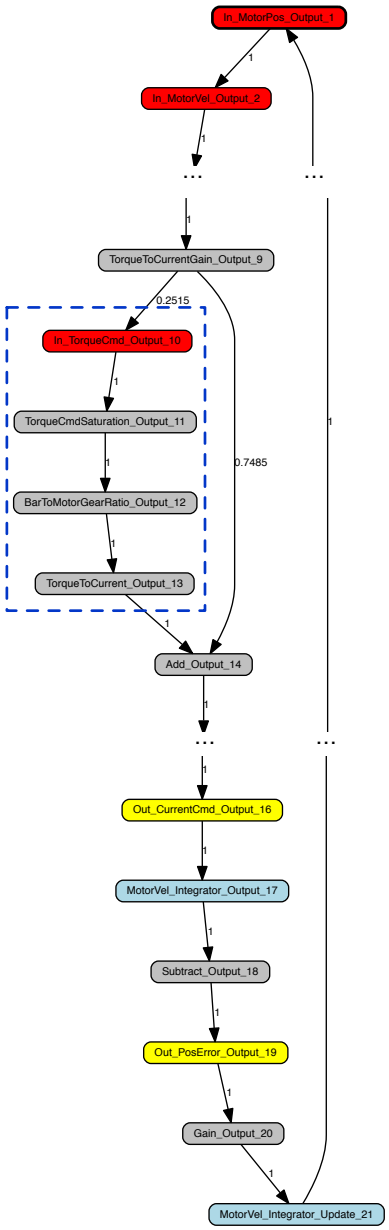

Figure 5: A control flow graph of the DEPM model, automatically generated from the reference Simulink model. Block functions of the Simulink blocks play the role of the nodes of the graph. The edges are weighted with transition probabilities describing the control flow between the block functions.

Non-virtual blocks are implemented with two types of block functions. Output functions compute output variables of the blocks. Update functions compute state variables of the blocks. Stateless blocks, e.g. Sum or Gain, are implemented with only one output block function. The blocks, which have states, like Integrator, are implemented with the both block functions. Our method transforms a block function into a DEPM element.

Designing a Simulink model, a developer specifies so-called block sample times that define how frequently block functions are executed. Simulink determines the sorted order in which to invoke the block functions. In each computation step, first all output functions and then all update functions are executed according to the sorted order and the sample times of the blocks.

The Simulink API allows running user-implemented callbacks before and after each output or update block function. We use this mechanism in order to instrument the model and gather the information about the execution order of the block functions. The run of the instrumented Simulink model gives an execution sequence of the block functions, which is transformed into the set control flow arcs of the DEPM model. The probabilities of control flow transitions are also computed using this sequence.

Figure 5 shows a control flow graph, generated automatically from the reference Simulink model. The elements E are the nodes and the control flow arcs A_{CF} are the edges of the control flow graph. The block *MotorVel Integrator*, highlighted in blue in Figure 2, has a state variable. Thus, it is implemented with both, output and update, block functions. The DEPM elements that represent these functions are also highlighted in Figures 5 and 6.

Four blocks on the bottom of the reference Simulink model in Figure 2 have sample times equal to 0.08, which differs from the sample times of the other blocks that is equal to 0.02. This means that the block functions of these four blocks are executed on each fourth iteration of the main loop. Therefore, we can observe control flow branching in the control flow graph, highlighted with the blue frame in Figure 5.

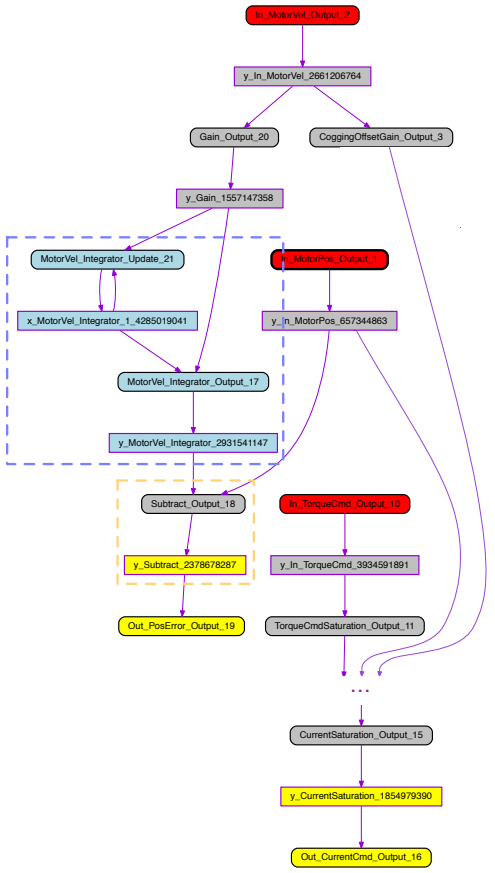

Figure 6: A data flow graph of the DEPM model automatically generated from the reference Simulink model. Block functions (rounded rectangles) and the output and state variables of the block functions (rectangles) are the nodes of the graph. The edges of the graph represent data flow.

5.2 Data flow analysis

Data flow analysis is the second step of the transformation method. The DEPM sets of data storages D and data flow arcs A_{DF} are generated in this step. In contrast to the control flow analysis, the data flow analysis is based on the static parsing of a Simulink model and does not require simulation.

Simulink blocks are interconnected with lines, which define signal flow between the output and input ports of the blocks. The Simulink API provides full access to the model, blocks, ports, and lines. The analysis of the data flow is realized as an algorithm that traces these lines.

Figure 6 shows a data flow graph, automatically generated from the reference Simulink model. The data storages D (rectangles) and the elements E (rounded rectangles) are the nodes and the data flow arcs A_{DF} are the edges of the data flow graph.

For each output port of each non-virtual block, we create a DEPM data storage and a data flow arc from the DEPM element that represents the output block function of this block to the created data storage. The names of these data storages are prefixed with 'y'.

For each state variable (if any) of each non-virtual block we create a DEPM data storage and two data flow arcs. The first data flow arc goes from the DEPM element that represents the update block function of this block to the created data storage. The second data flow arc goes in the opposite direction. The names of the created data storages are prefixed with 'x'.

For example, in Figure 6, in the region highlighted with the blue frame, the Simulink block *MotorVel Integrator* is represented by two block functions *MotorVel_Integrator_Output_17* and *MotorVel_Integrator_Update_21*. Its state variable is represented by the data storage *x_MotorVel_Integrator_1_4285019041* and the output variable by the data storage *y_MotorVel_Integrator_2931541147*. In the other hand, the Simulink block *Subtract* is stateless (highlighted with the yellow frame). It is implemented with only one output block function, represented by the DEPM element *Subtract_Output_18*. It has only one output variable, represented by the data storage *y_Subtract_2378678287*.

The outputs of the blocks are the inputs for the next blocks according to the signal flow. These connections are mapped to the DEPM using a recursive algorithm that traces

the lines of the Simulink model. It starts from each output port of each non-virtual block (source block) and goes, possibly through a number of virtual blocks like Muxes, Demuxes, Bus Creators, and Bus Selectors, until a first non-virtual block (target block) is met. After that, it creates new DEPM data flow arcs from the corresponded 'y'-data storage of the source block to the elements the represent block functions of the target block.

The implementation of the data flow analysis is quite complex because of heterogeneity of the Simulink blocks, the necessity to skip different types of virtual blocks and correctly process model hierarchy, implemented with virtual Subsystem blocks.

5.3 Block-level analysis

In the last step, the transformation method performs a number of error injection experiments in order to generate the set of DEPM conditions C. This is the longest step because it involves several individual simulations for each non-virtual Simulink block.

First, we instrument the Simulink model with callbacks in order to store all values of input, state, and output variables of each non-virtual block. After that, we create a simple Simulink model (local model) for each block. An example of a local model of the block Add of the reference Simulink model is shown in Figure 7. A local model contains a block itself, Constant blocks that are connected to the input ports of the block, and Terminator blocks that are connected to the output ports. Local models are also instrumented with callbacks in order pass the values from the stored data to the input and state variables and compare current values of output variables with the stored, correct values.

The number of simulations of a local model depends on the number of input ports of the block. For instance, we perform four simulations for the local model that is shown in Figure 7: First, we pass correct values to the both inputs, then we inject an error into the first input, into the second, and finally into the both inputs. During each simulation of a local model, we compute the number of errors for each output variable of an output block function, and for each state variable of an update block function.

```
...
<conditions element_name="BarToMotorGearRatio_Output_12">
 <if statement="y_TorqueCmdSaturation_652370398==ok">
   <then prob="1"
    update="y_BarToMotorGearRatio_601167163=ok"/>
 </if>
 <if statement="y_TorqueCmdSaturation_652370398==error">
  <then prob="0.02381"
   update="y_BarToMotorGearRatio_601167163=ok"/>
  <then prob="0.97619"
   update="y_BarToMotorGearRatio_601167163=error"/>
 </if>
</conditions>
...
<conditions element_name="In_TorqueCmd_Output_10">
 <if statement="True">
  <then prob="0.95"
   update="y_In_TorqueCmd_156773656=ok"/>
  <then prob="0.05"
   update="y_In_TorqueCmd_156773656=error"/>
 </if>
</conditions>
...
```

Listing 2: Examples of the generated conditions of DEPM elements, stored in the XML format.

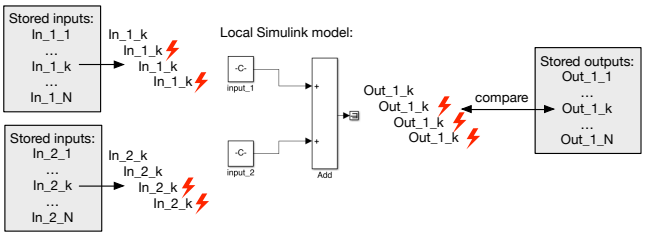

Figure 7: An example of an automatically generated local model for a block of the reference Simulink model. Reliability properties of DEPM elements are generated via error injection experiments with the local models.

This statistical information, gathered from the local models, is used for generation of the probabilistic conditions of the elements. For example, the first group of condition in Listing 2 is generated for the DEPM element *BarToMotorGearRatio_Output_12* that represents the output function of the Simulink block *BarToMotorGearRatio*. For the error-prone blocks, which are marked in red in the source Simulink model, we generate the condition using the provided probabilities, see the second group of conditions in Listing 2.

The errors are injected by the flipping of one random bit of a correct value. However, a single bit-flip does not always lead to the recognizable change of the value, especially for floating point data types. Therefore, we use a delta parameter, which was set to 1e-5 for the transformation of the reference Simulink model. If the difference between the original value and the value after a bit-flip is less than this delta then we flip another random bit. After the second "unsuccessful" random attempt, we flip 31st bit of a single or 63rd bit of a double.

5.4 Method application example

The DEPM XML file, generated according to the discussed method, has been analyzed with ErrorPro. The mean number of errors in the output blocks have been computed, see Table 1 - Analysis.

The method has been verified using our Simulink based error propagation simulator ErrorSim. 1000 simulations have been performed using the same error injection approach as in the discussed block-level analysis. During the simulations, the errors were injected into the input blocks with the defined probabilities. The errors detected in the output blocks have been counted, see Table 1 - Simulation.

Table 1: An example of the analytical results obtained after the application of the introduced transformation method: Mean numbers of errors in the output blocks computed using simulative and analytical approaches respectively.

Block name	Analysis	Simulation
Out_PosError	451.4	448.6
Out_CurrentCmd	35.35	34.9

The numbers are rather similar, the relative difference is about 1%. The similar difference has been observed for the other HAPTICS-2 models. The mean number of errors in the *Out_PosError* is much higher than in the *Out_Current-*

Cmd, because a single error, stored in the state variable of the *MotorVelIntegrator* block, will "infect" all the further outputs. This makes *In_MotorVel* the most critical input and *Out_PosError* the most sensitive output of the reference Simulink model.

5.5 Limitations

The transformation method supports only fixed-step discrete solvers. Continuous blocks are not supported. Sample times of all blocks must be explicitly defined. The following virtual blocks are not supported: Goto, Goto Tag Visibility, From, Enable, Signal Specification, and Trigger. Subsystems are always treated as virtual blocks.

The key limitations of the method are technical. SimPars, the current realization of the introduced method, is our first prototype. It has been tested with 11 real HAPTICS-2 models provided by ESA and several other small test models. The heterogeneity of the Simulink block set is the main future challenge.

Also, the current version has much room for performance optimization. The processing of the discussed reference model takes about 100 seconds on a commodity laptop (control flow analysis: 5s, data flow analysis: less than 1s, block-level analysis: 95s). The callback functions, which are used for the control flow and block-level analyses, are the heaviest part. The control flow analysis requires only one run of the instrumented model with fast callbacks that only store the control flow sequence. In contrast, the block-level analysis requires one run of the instrumented model with slow callbacks that store input and output values, and the runs of all instrumented local models.

Stateflow blocks are partially supported in the current version of SimPars. However, this is out of the scope of this article.

6. CONCLUSIONS

A new method for automatic transformation of Simulink models into DEPM models for further stochastic error propagation analysis has been introduced in this article. The method involves static parsing of a Simulink model as well as automatic model instrumentation and simulation. The method consists of three steps: control flow analysis, data flow analysis, and block-level analysis. A prototype of the transformation software tool has been implemented. The method has been applied to several Simulink models of a real space robotic project. The method was demonstrated as a part of stochastic analysis using one of these models.

7. ACKNOWLEDGMENTS

This research was supported by the European Space Agency (Project ESA-IPL-PTM-PA-gp-2014-817-LE). We thank our colleagues from the Telerobotics & Haptics at ESTEC who provided insight and expertise that greatly assisted the research. We also thank Regina Tuk for her assistance in the implementation of the transformation software prototype.

8. REFERENCES

[1] A. Agrawal, G. Simon, and G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[2] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. In *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*, pages 84–99, 2003.

[3] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 153–162, New York, NY, USA, 2003. ACM.

[4] K. Ding, T. Mutzke, A. Morozov, and K. Janschek. Automatic Transformation of UML System Models for Model-based Error Propagation Analysis of Mechatronic Systems. In *Accepted to Proceeding of 7th IFAC Symposium on Mechatronic Systems, Loughborough University, UK.*, 2016.

[5] ESA. METERON: Multi-Purpose End-To-End Robotic Operation Network. http://esa-telerobotics.net/meteron, 2014.

[6] ESA. HAPTICS-2. http://esa-telerobotics.net/meteron/flight-experiments/haptics-2, 2015.

[7] C. h. Yang and V. Vyatkin. Model transformation between MATLAB Simulink and Function Blocks. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1130–1135, July 2010.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[9] K. Janschek and A. Morozov. Dependability aspects of model-based systems design for mechatronic systems. In *Mechatronics (ICM), 2015 IEEE International Conference on*, pages 15–22, March 2015.

[10] MathWorks. IEC 61508 Support in Matlab and Simulink. http://www.mathworks.de/automotive/standards/iec-61508.html.

[11] MathWorks. IEC Certification Kit. http://www.mathworks.de/products/iec-61508/.

[12] A. Morozov and K. Janschek. Dual graph error propagation model for mechatronic system analysis. In *18th IFAC World Congress, Milano, Italy*, pages 9893–9898, 2011.

[13] A. Morozov and K. Janschek. Case study results for probabilistic error propagation analysis of a mechatronic system. *Tagungsband Fachtagung Mechatronik*, pages 229–234, 2013.

[14] A. Morozov and K. Janschek. Probabilistic error propagation model for mechatronic systems. *Mechatronics*, 24(8):1189 – 1202, 2014.

[15] A. Morozov, R. Tuk, and K. Janschek. ErrorPro: Software Tool for Stochastic Error Propagation Analysis. In *1st International Workshop on Resiliency in Embedded Electronic Systems, Amsterdam, The Netherlands*, pages 59–60, 2015.

[16] OMG. OMG Unified Modeling Language (OMG UML), 2015.

[17] B. Panzer-Steindel. Data integrity. *CERN/IT*, 2007.

[18] D. Ramos-Hernandez, P. Fleming, and J. Bass. A novel object-oriented environment for distributed process control systems. *Control Engineering Practice*, 13(2):213 – 230, 2005.

[19] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, USA, 2009. ACM.

[20] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional Translation of Simulink Models into Synchronous BIP. In *IEEE Fifth International Symposium on Industrial Embedded Systems*, pages 217–220, Trento, Italy, July 2010. IEEE.

[21] C.-J. Sjöstedt, J. Shi, M. Törngren, D. Servat, D. Chen, V. Ahlsten, and H. Lönn. Mapping Simulink to UML in the Design of Embedded Systems: Investigating Scenarios and Structural and Behavioral Mapping. In *OMER 4 Post Workshop Proceedings*, Apr. 2008.

[22] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to Lustre. *ACM Trans. Embedded Comput. Syst.*, 4(4):779–818, 2005.

[23] V. Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. ISA, 2007.

[24] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics (1978-1994). *IBM J. Res. Dev.*, 40(1):3–18, Jan. 1996.