IMPROVED STOCHASTIC CONTROL FLOW MODEL FOR LLVM-BASED SOFTWARE RELIABILITY ANALYSIS

V. Vidineev[1], N. Yusupova[1], K. Ding, A. Morozov[2], K. Janschek[2]
Faculty of Computer Science and Robotics, Ufa State Aviation Technical University, Ufa, Russia[1]
Technische Universität Dresden, Germany[2]
vidineev.vyachesla01@ugatu.ac.ru, yussupova@ugatu.ac.ru, kai.ding tu-dresden.de, andrey.morozov tu-dresden.de, klaus.janschek@tu-dresden.de

Abstract: *Recently we have proposed a new method for error propagation analysis of the safety-critical software using the transformation of the source code to the Dual-graph Error Propagation Model (DEPM) based on the Low-Level Virtual Machine (LLVM) compiler framework, that allows the automatic analysis of C-code or another LLVM supported front-end. The source code is compiled into the LLVM Intermediate Representation and instrumented in order to analyze control and data flow structures of the software and the control flow transition probabilities between the basic blocks. Based on this information a DEPM for further analysis is generated. The DEPM is a stochastic model that captures system properties relevant to error propagation processes such as control and data flow structures and reliability characteristics of single components, LLVM instructions in this particular case. The DEPM helps to estimate the impact of a fault in a particular instruction on the overall system reliability, e.g. to compute the mean number of erroneous values in a critical system output during given operation time. The feasibility of the method has been proven on several case studies and also reveals several limitations of the current control flow model.*
This paper address the improvement of the control flow model using a new customizable heuristic method for the analysis of control flow sequences and their mapping into discrete-time Markov chain models. The method is designed in a way to keep a required tradeoff between the model size and precision.

1. Introduction

Analysis of the software reliability is an important part of the system-level dependability evaluation for any safety-critical industrial domain. We are working on a comprehensive stochastic evaluation of the propagation of data errors through complex software structures. We consider the faults can be described stochastically, focusing on bit flips and other computing hardware or network originated faults.

The evaluation of the fault probabilities is left out of the scope of our research focus. Examples of the studies for space and automotive domains can be found in [1, 2].

The main goal is the evaluation of the probability that a data error will reach a critical system output given the fault activation probabilities and the system operation time. The mathematical model behind our analytical approach is called Dual-graph Error Propagation Model (DEPM) that will be discussed in more details in Section 2.1. We are putting effort on automatic generation of these models from the source code. An overview of our LLVM-based DEPM generation approach will be given in Section 2.2.

This paper introduces a new, improved algorithm for the generation of stochastic control flow model for the DEPM. The method is designed in a way to keep a required tradeoff between the model size and precision. Section 3 gives a reference example that will be used in Section 4 that in turn describes the proposed algorithm.

Fig. 1 shows a simple DEPM example. An *element*, e.g. *A*, *B*, or *C*, represents a fundamental executable part of a system. Each element may receive input data and provide output data. A *data storage*, e.g. *d1*, *d2*, or *d3*, represents a variable that can be read or written by an *element*. For instance, the *element C* reads *d2* and *d3*, and writes output. A *control flow edge* (black lines in Fig. 1), weighted with an attribute probability, represents a control flow transition between the *elements*. For instance, after the execution of *A*, *B* will be executed with the probability *0.7* and *C* with the probability *0.3*. A *data flow edge* (purple lines in Fig. 1) describes data transfer between the *elements*. A *data flow* connects an *element* with a *data storage* or vice versa. The *data flow* edges are considered to be the paths of the data error propagation. Fault activation and the error propagation are specified using probabilistic *conditions* of the *elements*, see the conditions of *A*, *B*, and *C* in the right part of Fig. 1.

2. State of the Art

2.1. Dual-graph Error Propagation Model

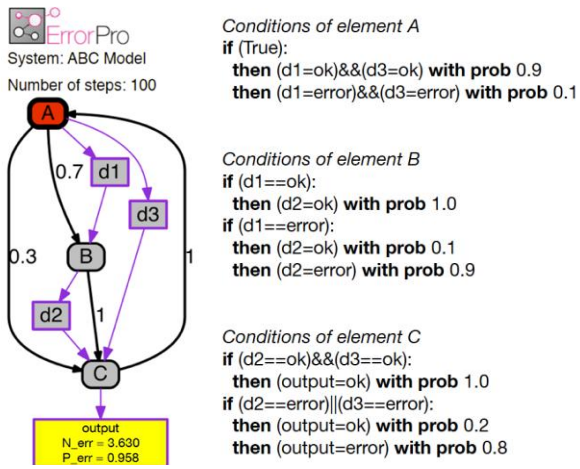

Fig 1. *A simple DEPM example and the conditions of the elements [4].*

A DEPM [3-5] is a mathematical model that captures control and data flow aspects of a system, described using the following set-based notation:

$$DEPM := < E, D, A_{cf}, A_{df}, C >$$

Where, E is a non-empty set of executable system 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, and C is a set of conditions of the elements.

During the execution of an element, faults can be activated and occurred errors propagate to its output data. For instance, in the *element A*, faults can be activated with probability *0.1*, defined in the conditions of *A* (see Fig. 1), and occurred errors propagate to its output data *d1* and *d3*. The error propagation probabilities for each element are defined also using probabilistic *conditions*. The errors can propagate from the inputs to the outputs. For instance, the *conditions* of the *element B* specify that the *element B* does not activate faults, but the errors can propagate from *d1* to *d2* with the probability *0.9*.

The DEPM allows the computation of several reliability metrics, such as the mean number of errors (N_{err}) and probability of errors

(P_{err}) in selected data storages. N_{err} stands for the average number of erroneous values in a data storage, and P_{err} is the probability of an error in a data storage during the system execution. For instance, the evaluated N_{err} in the data storage output during *100* steps (execution of one element is one step) is equal to *3.630*, and the P_{err} is *0.958*, as shown in Fig. 1. The computed reliability metrics are important measures for the system analysis, particularly for the reliability assessment and should comply with system requirements.

2.2. Generation of DEPM from source code

The DEPM generation method is based on the Low-Level Virtual Machine (LLVM) compiler framework, that allows the automatic transformation of C-code or another LLVM supported front-end into a DEPM. The source code is compiled into the LLVM Intermediate Representation and instrumented in order to analyse control and data flow structures of LLVM instructions and control flow transition probabilities. The obtained information is transformed into the formal DEPM XML for further analysis.

Fig. 2 shows the architecture of the proposed transformation method. Rounded rectangles with blue borders represent activities and grey rectangles represent data. Two LLVM passes have been developed as well as the python script that processes the outputs of the passes and generates DEPM XML file for further analysis with our tool OpenErrorPro [7]. All the steps are automated and can be run with a single shell script that calls LLVM tools (compilers, linkers etc), as well as our passes, and the DEPM generation python script. The detailed technical description of the transformation method is given in [6]. Here we would like to focus on the control flow part because it is relevant for the contribution of this paper.

The transformation pass, shown in Fig. 2, helps to identify elements of the future DEPM, control flow structure, and control flow transition probabilities. The pass takes the LLVM IR of the original C-Code generated with *clang* as input and performs two tasks:

(1) The first task is to generate the list of DEPM elements that represent single LLVM IR instructions. Using the LLVM API, the pass iterates over all LLVM IR instructions and stores the information into the *elements.txt* file. The pass gives a unique name for each instruction taking into account its location according to the LLVM IR hierarchy.

(2) The second task is the generation of the instructions execution sequence in order to define the structure of the DEPM control flow as well as control flow transition probabilities. In order to achieve this, we link an external C-code of a *"print"* function with the original LLVM IR using *llvm-link* tool and append calls to this function after each instruction of the original LLVM IR. After this instrumentation, we compile and run the program. The instructions execution sequence is stored in *sequence.txt*.

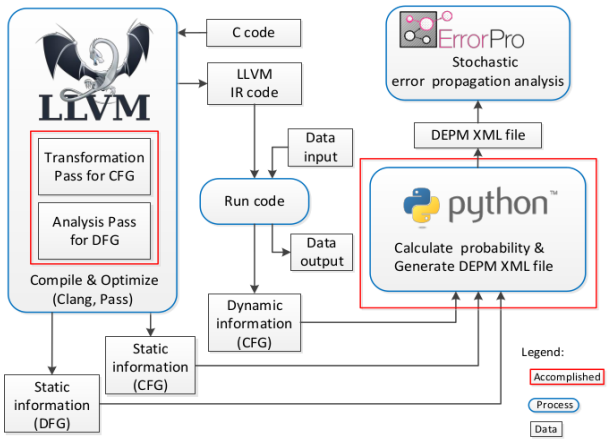

Fig 2. *Top-level architecture of the transformation method from C code to DEPM [6].*

After that the python script processes this sequence and in order to generate the DEPM control flow graph. In our original version, all control flow transitions are considered to be stochastic. We just count the number of control flow transitions between the elements and transform this information into control flow arcs and their transition probabilities.

However, this method has several drawbacks. This control flow generation method might cause a problem of discrepancy between the generated probabilistic control flow graph and the original deterministic sequence. Some parts of the original sequence can have a strong visible pattern that will be lost after the mapping of the sequence into the control flow graph that follows the Markov property and the control flow probabilities depend solely on the current element and do not take the execution history into consideration.

2.3. Serial test

The proposed solution in this paper includes the testing of the generated sequence for randomness. This can be done using the serial test method that is commonly used for testing random number generators. It is based on Chi-squared statistic to compare adjacent pairs or triples of numbers.

For instance, the serial test method is provided by the python library skidmarks [8], the detailed algorithm description is given in [9]. Listing 1 shows two examples of more and less random sequences:

```
>>> serial_test('101010101111000')
{'chi':1.4285714285714286,'p':0.6988513076924
8427}
>>>
serial_test('11000000000000011111111111')
{'chi':18.615384615384617,'p':0.0003283102182
6061683}
```

Listing 1. Examples of the skidmarks serial test on two binary sequences

To make a decision about randomness of the sequence we have to take a look at the p-values of the test. If the p-value is less than a defined constant the sequence is considered to be random.

3. Reference example

Let's consider the next abstract example in order to explain the main idea of the proposed methods for control flow analysis. Suppose that our reference piece of software operates according to the control flow graph that is shown in Fig. 3.

Assume that the discussed method for control flow analysis generated the following execution sequence:

$$ABE\ ABE\ ABE\ ACDE\ ABE\ ACE\ ABE\ ACDE\ ABE\ ABE\ ACE\ \ldots$$

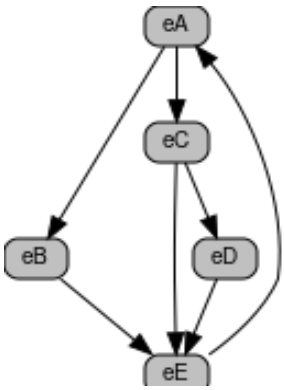

Fig 3. *The reference DEPM control flow graph.*

In this sequence there are two elements, which have two or more output control flow arcs: namely A and C. In the transitions from A to B or C we observe no regularity, while in the transitions from C to D or E we can see that it happens not by chance but according to a clear pattern:

$$CD, CE, CD, CE, CD, CE, \dots$$

The results are confirmed by the serial test shown in Listing 2:

```
>>>
serial_test('BBBCBCBCBBCCCBCCCBCCCBCBBBCCCCBC
BBCBC')

{'p':        0.6695087450687482,        'chi':
1.5555555555555556}

>>>
serial_test('DEDEDEDEDEDEDEDEDEDEDEDEDEDEDE
DEDED')

{'p': 1.0, 'chi': 0.0}
```

Listing 2. Skidmarks serial test results for the reference example.

So, we can consider the control flow transitions after the element A stochastic and the transition after the element B deterministic.

4. Algorithm description

Generalizing these results, we can define the next steps:
Step 1: Build a control flow graph based on the original sequence of execution of elements.

Step 2: Find elements that have two or more output arcs.

Step 3: For each such element build a sequence of elements which will be executed after this element.

Step 4: Check each of these sequences for randomness using the Serial test.

Step 5: For each sequence that is considered to be random statistically estimate control flow transitions probabilities based on the number of the actual transitions in the original sequence.

Step 6: For each not random sequence add a data storage with the rules that describe the uncovered control flow pattern.

5. Results of the algorithm application

The result of this algorithm applied to the reference example is shown in Fig 4. In this DEPM, the element *A* has purely stochastic control flow transitions: we jump to the element *eB* with the probability *0.63* and to the element *eC* with the probability *0.37*.

In the other hand, the element *C* contains the following conditional, guarded by the value of the data storage *i*, control flow transitions:

1) (i=0) => (cf'=eD) & (i'=i+1)

2) (i=1)=>(cf'=eE) & (i'=0)

That define the following control flow behavior. If the value of the data storage *i* is *0* that we jump to *eD* and increase the value of *i*. If the value of *i* is *1* then we jump to *eE* and reset *i* to its initial value *0*.

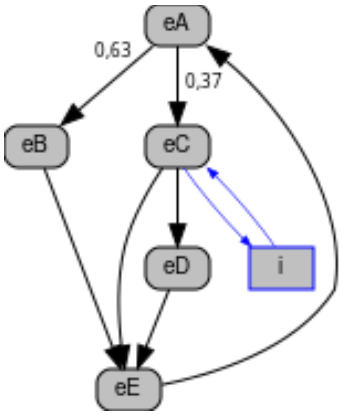

Fig 4. *The resulting DEPM control flow graph generated using the proposed algorithm.*

4. Conclusion

The data error propagation analysis is an important part of the reliability evaluation of safety critical software. In our recent works we have proposed a stochastic Dual-graph Error Propagation Model (DEPM) and an automatic method for the generation of the DEPM for the software using LLVM technology. In this paper, a new algorithm for the automatic generation of the DEPM control flow graph has been introduced. The algorithm is based on the statistical evaluation of the execution sequence. The key feature of the new algorithm that it can distinguish between stochastic and deterministic control flow transitions via the application of the serial test for randomness.

References

1. I.Verzola, A.E.Lagny, and J.Biswas, "A predictive approach to failure estimation and identification for space systems operations," SpaceOps 2014 (Pasadena, CA), 2014.
2. P. Koopman, "A case study of toyota unintended acceleration and software safety," *Presentation. Sept*, 2014.
3. A. Morozov, Dual-graph Model for Error Propagation Analysis of Mechatronic Systems. Dresden: Jörg Vogt Verlag, 2012.
4. A. Morozov and K. Janschek, "Probabilistic error propagation model for mechatronic systems," Mechatronics, vol. 24, no. 8, pp. 1189 – 1202, 2014.
5. K. Ding, T. Mutzke, A. Morozov, and K. Janschek, "Automatic transformation of uml system models for model-based error propagation analysis of mechatronic systems," IFAC-PapersOnLine, vol. 49, no. 21, pp. 439–446, 2016.
6. A. Morozov, Y. Zhou, K. Janschek, "LLVM-based Stochastic Error Propagation Analysis of Manually Developed Software Components", Proceedings of European Safety and Reliability Conference (ESREL). Trondheim, Norway, 2018.
7. OpenErrorPro on the github. https://mbsa-tud.github.io/OpenErrorPro/, 2018
8. Skidmarks library for randomness serial test https://pypi.org/project/skidmarks/, 2018
9. Morgan, Byron JT. *Elements of simulation*. Vol. 4. CRC Press, 1984